Toward harnessing a Java high-level language virtual machine for supporting software testing

Vinicius Humberto Serapilha Durelli

Utilizando uma máquina virtual Java como apoio à atividade de teste de software

Vinicius Humberto Serapilha Durelli

Toward harnessing a Java high-level language virtual machine for supporting software testing

Vinicius Humberto Serapilha Durelli

Advisor: **Prof. Dr. Marcio Eduardo Delamaro**
Co-advisor: **Prof. Dr. Jeff Offutt**

Doctoral dissertation submitted to the Instituto de Ciências Matemáticas e de Computação - ICMC-USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION.*

USP – São Carlos
November 2013

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

```
       Durelli, Vinicius Humberto Serapilha
D955t     Toward harnessing a Java high-level language
       virtual machine for supporting software testing /
       Vinicius Humberto Serapilha Durelli; orientador
       Marcio Eduardo Delamaro; co-orientador Jeff Offutt.
       -- São Carlos, 2013.
          165 p.


          Tese (Doutorado - Programa de Pós-Graduação em
       Ciências de Computação e Matemática Computacional) --
        Instituto de Ciências Matemáticas e de Computação,
       Universidade de São Paulo, 2013.


          1. software testing. 2. mutation testing. 3.
       weak mutation. 4. record-and-playback mechanism. 5.
       Java virtual machine. I. Delamaro, Marcio Eduardo,
       orient. II. Offutt, Jeff, co-orient. III. Título.
```

Utilizando uma máquina virtual Java como apoio à atividade de teste de software

Vinicius Humberto Serapilha Durelli

Orientador: **Prof. Dr. Marcio Eduardo Delamaro**
Co-orientador: **Prof. Dr. Jeff Offutt**

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Matemática . *VERSÃO REVISADA*

USP – São Carlos
Novembro de 2013

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados fornecidos pelo(a) autor(a)

```
       Durelli, Vinicius Humberto Serapilha
D955t     Toward harnessing a Java high-level language
       virtual machine for supporting software testing /
       Vinicius Humberto Serapilha Durelli; orientador
       Marcio Eduardo Delamaro; co-orientador Jeff Offutt.
       -- São Carlos, 2013.
          165 p.


          Tese (Doutorado - Programa de Pós-Graduação em
       Matemática) -- Instituto de Ciências Matemáticas e
       de Computação, Universidade de São Paulo, 2013.


          1. teste de software. 2. teste de mutação. 3.
       mutação fraca. 4. mecanismo de record-and-playback.
       5. máquina virtual Java. I. Delamaro, Marcio
       Eduardo, orient. II. Offutt, Jeff, co-orient. III.
       Título.
```

I Want Out

From our lives' beginning on
We are pushed in little forms
No one asks us how we like to be
In school they teach you what to think
But everyone says different things
But they're all convinced that
They're the ones to see

So they keep talking and they never stop
And at a certain point you give it up
So the only thing that's left to think is this

I want out – to live my life alone
I want out – leave me be
I want out – to do things on my own
I want out – to live my life and to be free

People tell me A and B
They tell me how I have to see
Things that I have seen already clear
So they push me then from side to side
They're pushing me from black to white
They're pushing 'til there's nothing more to hear

But don't push me to the maximum
Shut your mouth and take it home
'Cause I decide the way things gonna be

I want out – to live my life alone
I want out – leave me be
I want out – to do things on my own
I want out – to live my life and to be free

There's a million ways to see the things in life
A million ways to be the fool
In the end of it, none of us is right
Sometimes we need to be alone

No no no, leave me alone

I want out – to live my life alone
I want out – leave me be
I want out – to do things on my own
I want out – to live my life and to be free

Helloween, Keeper of the Seven Keys, Pt. 2 (1988)

Acknowledgments

To quote the Grateful Dead, *"What a long strange trip it's been"*. Looking back on the past few years, I realized that along the way I had tremendous help from friends and colleagues too numerous to name them all in this document (so, any omission here is not intentional). I also received extensive support from my family, international researchers, and funding agencies. So, I would like to acknowledge all the people that have turned my last few years into a great experience.

Foremost, I would like to express my deepest gratitude to my friend Simone Borges for placing food in front of me in order to keep me alive while I was trying to meet deadlines. Simone, you did your best to keep me going through the toughest times, and for that I am immensely thankful. Simone should also be thanked for her help with most of the figures in this document, without her they would not be nearly as polished. Simone also kept reminding me that there is more to life than work: we played Zelda, Super Smash Bros. Brawl, and Super Mario Galaxy whenever we had a couple hours of leeway. I had fun playing video games with you, those were great times filled with laughter. And speaking of playing video games, I also would like to thank my brother (Rafael Durelli) for always being there for me, and for playing Super Smash Bros. Brawl with Simone and me (as far as Super Smash Bros. Brawl is concerned, the more players the better).

I am thankful to Dr. Márcio Delamaro for his invaluable feedback and guidance throughout the past few years. It is utterly my fault if his high standards are not reflected in this document.

My sincere thanks also goes to Dr. Jeff Offutt. During my period abroad I learned a lot from him. Our chats helped me to figure out how I should go about doing research and writing papers — yet, it is probably safe to say that my writing will never rival the succinctness of his style.

I am thankful to all core members of the Maxine VM project at Oracle Laboratories. In particular, Douglas Simon for his always enlightening and accurate answers on the Maxine

VM mailing list. Thanks to the emails exchanged with him, I was able to tease out the chunks of code I had to study and change. Further, whenever I was at a loss as to why my code was acting up, he provided me with helpful troubleshooting suggestions.

Grateful thanks are due to Nan Li. I am glad we became friends. I enjoyed spending most Sunday mornings playing soccer with you. Also, it goes without saying that being one of the groomsmen at your wedding meant a lot to me.

For the sake of finishing my Ph.D., I took many friends for granted. Still, they have not given up on me. Besides friends, some colleagues and fellow researchers that I met along the way have contributed to make the last few years even more special. The list of friends, colleagues, and fellow researchers is the following (I hope I did not leave anyone off the list): Rodrigo Fraxino, André Endo, Fabiano Ferrari, Lucas Bueno, Jorge Cutigi, Kátia Felizardo, Bruno Cafeo, Marco Silva, Aretha Alencar, Wesley Renan, Upsorn Praphamontripong, Rafael Oliveira, Sunitha Thummala, Paulo Nardi, Rosana Braga, Rosângela Penteado, Matheus Viana, Marcos Laia, Marcelo Eler, Ricardo Rios, and Tatiane Rios.

I am immensely grateful to my mother (Lúcia Serapilha) and grandmother (Glinauria Serapilha). I know that they will never want to read this document but they contributed to it being written. Thank you both for your unconditional support and love.

This research was made possible by the financial support provided by FAPESP (grant number 2009/00632-1) and CAPES (grant number 0340-11-1).

Abstract

High-level language virtual machines (HLL VMs) have been playing a key role as a mechanism for implementing programming languages. Languages that run on these execution environments have many advantages over languages that are compiled to native code. These advantages have led HLL VMs to gain broad acceptance in both academy and industry. However, much of the research in this area has been devoted to boosting the performance of these execution environments. Few efforts have attempted to introduce features that automate or facilitate some software engineering activities, including software testing. This research argues that HLL VMs provide a reasonable basis for building an integrated software testing environment. To this end, two software testing features that build on the characteristics of a Java virtual machine (JVM) were devised. The purpose of the first feature is to automate weak mutation. Augmented with mutation support, the chosen JVM achieved speedups of as much as 95% in comparison to a strong mutation tool. To support the testing of concurrent programs, the second feature is concerned with enabling the deterministic re-execution of Java programs and exploration of new scheduling sequences.

Keywords — software testing; mutation testing; weak mutation; record-and-playback mechanism; Maxine VM; Java virtual machine.

Resumo

MÁQUINAS virtuais de linguagens de programação têm desempenhado um papel importante como mecanismo para a implementação de linguagens de programação. Linguagens voltadas para esses ambientes de execução possuem várias vantagens em relação às linguagens compiladas. Essas vantagens fizeram com que tais ambientes de execução se tornassem amplamente utilizados pela indústria e academia. Entretanto, a maioria dos estudos nessa área têm se dedicado a aprimorar o desempenho desses ambientes de execução e poucos têm enfocado o desenvolvimento de funcionalidades que automatizem ou facilitem a condução de atividades de engenharia de software, incluindo atividades de teste de software. Este trabalho apresenta indícios de que máquinas virtuais de linguagens de programação podem apoiar a criação de ambientes de teste de software integrado. Para tal, duas funcionalidades que tiram proveito das características de uma máquina virtual Java foram desenvolvidas. O propósito da primeira funcionalidade é automatizar a condução de atividades de mutação fraca. Após a implementação de tal funcionalidade na máquina virtual Java selecionada, observou-se um desempenho até 95% melhor em relação a uma ferramenta de mutação forte. A fim de apoiar o teste de programas concorrentes, a segunda funcionalidade permite reexecutá-los de forma determinística além de automatizar a exploração de que novas sequências de escalonamento.

Palavras-chave — teste de software; teste de mutação; mutação fraca; mecanismo de record-and-playback; Maxine VM; máquina virtual Java.

Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter

1

Introduction

High-level language virtual machines (HLL VMs) have been playing an important role in the implementation of high-level languages (HLLs) for more than forty years (Craig, 2005). Over the years, a number of prominent HLLs have been designed to run on these managed execution environments. The HLL VMs for the languages Lisp (Steele and Gabriel, 1993), Pascal (Wirth, 1996), and Smalltalk (Goldberg and Robson, 1983; Kay, 1996) can be regarded as early implementations that fostered and popularized the concept. These early HLL VMs pioneered a number of innovative techniques, e.g., incorporation of runtime[1] compilers (Deutsch and Schiffman, 1984; Arnold et al., 2005b; D'Hondt, 2008) and automatic reclamation of storage (Lo et al., 2003; Jones et al., 2011). Thus, it can be argued that they set the standard for later implementations.

Currently, the Java virtual machine (JVM) (Li, 1998; Lindholm and Yellin, 1999; Engel, 1999) and the Common Language Runtime (CLR) (Meijer and Gough, 2012) are arguably the most widely used HLL VMs and have been driving such technology into the mainstream (Smith and Nair, 2005b; Durelli et al., 2010). Other prominent contemporary examples are the HLL VMs for the languages Ruby (Sasada, 2005; Flanagan and Matsumoto, 2008; Thomas et al., 2009), Scala (Wampler and Payne, 2009; Odersky et al., 2011),

[1]Following the convention proposed by Smith and Nair (2005b), throughout this document the single-word form *runtime* refers to the virtualizing runtime software in a virtual machine, whereas *run time* and *run-time* are used in a less specific sense: the time during which a program is running and the amount of time it takes to execute a certain program, respectively.

Perl (Schwartz et al., 2011), Python (Lutz, 2009), and Erlang (Cesarini and Thompson, 2009).

The main benefit provided by the early HLL VM implementations was cross-platform portability. In the following years, however, HLL VMs have become the *de facto* standard for implementing programming languages largely due to the other software engineering benefits they have brought to the mainstream, e.g., program isolation, built-in multithread support, and automatic memory management (Craig, 2005; Smith and Nair, 2005b). Likewise, some of these intrinsic advantages over statically compiled binaries have led to their widespread adoption on various platforms as real-time (Higuera-Toledano et al., 2000; Cavanagh and Wang, 2005; Baker et al., 2006; Armbruster et al., 2007; Higuera-Toledano, 2012), low-end embedded systems (Chen and Kandemir, 2005; Koshy et al., 2009), and mobile devices (Lawton, 2002; Riggs et al., 2003; Wolfe, 2004).

According to Ryder et al. (2005), software engineering research and practice have been enjoying a symbiotic relationship with language design. For instance, modules can be seen as an outgrowth of concepts such as information hiding and modularization. Similarly, data abstraction led to the development of object-oriented languages, and the widely adoption of such languages can be ascribed to modularity and reliability, which are established themes in software engineering research (Ryder and Soffa, 2003). Moreover, many languages have been devised to cope with unique demands of certain applications (Sammet, 1972). However, the influence that research in software engineering has over programming language design has mainly led to improvements at language-level.

Although HLL VMs have become widely used for implementing HLLs, most research in this area has been concerned with boosting the performance of such managed execution environments. In fact, as pointed out by a systematic mapping on the topic (Durelli et al., 2010), the bulk of the research on HLL VMs has focused on coming up with increasingly faster adaptive optimization techniques (Davis and Waldron, 2003; Arnold et al., 2005b) and more effective memory management algorithms (Kazi et al., 2000). Few efforts have tried to exploit the intrinsic control that HLL VMs exert over running programs to support and speed up software engineering activities.

This research suggests that software testing is one of the software engineering activities that can capitalize on HLL VMs support. We followed the threefold formula proposed by Booth et al. (2008) to flesh out the scope of this research:[2]

- **Research Problem:** Research in HLL VMs is primarily concerned with devising and implementing optimization and memory management algorithms. There has

[2] Booth et al. refer to research problem as *practical* or *conceptual problem* and potential contribution as *significance*.

been very little research on investigating how execution environments can be used to automate software engineering activities as software testing. As an initial step towards addressing this lack of knowledge, this research is aimed at advancing the knowledge and understanding of whether HLL VMs provide a viable option for supporting software testing.

- **Research Questions (RQs):**

  - $\mathbf{RQ}_1$: Can HLL VMs be harnessed to support software testing?

  As the results of retrofitting software testing support into an HLL VM become increasingly apparent, we intend to fine-tune the previous question by asking:

  - $\mathbf{RQ}_2$: What sort of software testing support is more suited to modern HLL VMs?

- **Potential Contribution:** We are interested in understanding how HLL VMs can be harnessed to support software testing. We conjecture that some software testing activities that are currently implemented by heavily relying on instrumentation or libraries should rather be implemented at HLL VM level. In other words, this research can be seen as an initial foray into understanding how the interplay between the execution environment and programs can be explored for software testing purposes. Thus, *conceptual consequences* (Booth et al., 2008) of this investigation will help researchers and practitioners to gain a deeper understanding of which characteristics of the contemporary HLL VMs can be extended for the purpose of automating software testing techniques. Given that we set out to modify a full-fledged HLL VM to automate two software testing activities (described in the following sections), the resulting HLL VM can be seen as a *practical consequence* (Booth et al., 2008) of this research.

1.1 Retrofitting Software Testing Support into HLL VMs

In order to investigate whether retrofitting software testing support into managed execution environments has a positive impact on testing activities, we decided to tackle a complex problem: setting up an infrastructure to support the testing of concurrent programs using mutation testing. Towards this end, the following features were identified as essential: *(i)* automating the execution of mutants, *(ii)* enforcing the deterministic re-execution of multithreaded programs, and *(iii)* enabling that subsequent executions cover schedules different from the ones previously explored.

We decided to focus on mutation testing because, despite being a mature technique, the cost associated with mutants execution is still one of the main challenges faced by researchers and practitioners. In addition, we argue that the problem is worth tackling because previous studies have been restricted to modifying compilers (Untch et al., 1993) and interpreters (Offutt and King, 1987), no one has investigated whether HLL VMs can be of assistance in speeding up the execution of mutants.

The second and third features can be useful when used as part of a much larger testing environment encompassing more testing techniques. They play a supportive role to debugging and testing techniques in general. Although mutation and the other two features may seem unrelated, they are complementary. For instance, these three features can be combined to create a more sophisticated testing environment that allows for applying mutation testing to concurrent programs. The next two subsections detail these features.

1.1.1 Harnessing HLL VMs for Speeding up Mutation Testing

Activities that entail investigating the dynamic behavior of executing programs, in particular some software testing activities, can benefit greatly from HLL VMs support. One of these activities is mutation analysis (DeMillo et al., 1978; Jia and Harman, 2011).

Generally, mutation analysis concepts are implicitly implemented atop HLL VMs, which may hinder managed execution environments in performing their optimizations. We posit that mutation analysis can be further sped up by having its main concepts (e.g., mutants) supported as first-class status (Scott, 2009) within HLL VMs. Moreover, by building mutation analysis support into an HLL VM implementation it is possible to achieve extensive control over program execution; much greater than the one available to tools built atop of execution environments.

When performing traditional mutation analysis, a significant portion of code that has already been executed (during the execution of the original program) needs to be repeatedly executed again for each mutant. That happens because each mutant usually contains only one changed statement, thus a mutant execution is identical to the original program up to the point at which the mutated statement is run. Such an execution approach, that executes each mutant from the beginning, results in unnecessary computational burden.

Our functionality consists in further speeding up mutation analysis by making the target HLL VM execute the original program and its respective mutants in a more effective fashion, which entails avoiding re-executing large chunks of code that are common to both versions of the program. This can be achieved in the following way: the mutation-aware HLL VM has to execute the original program up to the point where the first mutated statement occurs, from that location on, it triggers the execution of mutants. Upon completion of a mutant, the HLL VM compares the result of the original program to the one produced by the mutant, deciding whether such mutant should be marked as dead. After executing and comparing the results of all mutants, the execution of the original program resumes. The execution is forked whenever the HLL VM executes chunks of the original code which have corresponding mutated versions. Such execution approach is along the lines of the approach called split-stream, which was first envisioned by King and Offutt (1991) as an improvement to their Fortran language system for mutation-based software testing, but not implemented.

One of the advantages of the split-stream approach is that it requires only one execution per test case to evaluate all mutants, hence only one instance of the underlying HLL VM is needed. This is in stark contrast to the traditional approach in which there is a one-to-one relationship between the HLL VM and the program under test. Consequently, the conventional approach demands n instances of the HLL VM to execute n mutants.

The proposed functionality aims to take the split-stream approach even further by capitalizing on the sophisticated multithread support provided by modern HLL VMs to run each mutant in its own thread. Concurrently executing mutants and comparing their results after running their respective original counterparts makes this a weak mutation approach. The term weak is used in the sense that this sort of mutation only requires the intermediate values to be different, the mutated program may still yield the same output as the original. So, in essence, the term has to do with the way mutants are compared, and not with effectiveness.

1.1.2 Towards Predictably Recording and Replaying Multithreaded Programs and Progressively Exploring Interleavings

The increasing momentum of multicore computers has been pushing concurrent languages into the mainstream. However, the performance advantages of concurrency come at the cost of increased complexity. Reasoning about, implementing, testing, and debugging concurrent programs are complex activities (Gatlin, 2004; Sutter and Larus, 2005; Lu et al., 2008), most of this complexity stems from the inherent nondeterministic behavior of concurrent programs.

Many languages have made great strides towards making concurrent programs easier to implement, exposing software practitioners to accessible concurrent programming constructs and libraries that further support concurrency (Larson, 2008, 2009). Nevertheless, no high-level programming language provides a way to deterministically control which schedules will take place each time a program executes.

During run time, thread-management decisions are made by the underlying scheduler, making it difficult to evaluate whether a program behaves as expected on every possible thread schedule. This nondeterminism renders conventional testing and debugging impractical (Carver and Tai, 1991). Often, when conventional testing is carried out with no additional strategy to either control or tamper with execution, only minor variations of the same thread interleaving tend to be exercised (Wang et al., 2011). To make matters worse, there are also concurrency problems that tend to disappear or behave slightly differently when one attempts to isolate them. These hard-to-diagnose problems, called *Heisenbugs* (Grötker et al., 2012), are rather difficult to reproduce during testing (Ball et al., 2011). Consequently, many concurrency problems manifest themselves only when the programs are already in production.

In effect, even a common practice such as stress testing is not very effective to uncover these insidious, error-yielding problems. Simply put, executing a given program multiple times does not ensure that a fault-manifesting schedule will turn up. Given that, automated support is imperative when dealing with concurrent programs. An approach to cope with the unpredictability of concurrent programs during testing and debugging is to implement tools that make it possible to capture information about a given execution. Then, in a later stage, these tools use the acquired information to deterministically enforce the re-execution of the previously recorded execution.

Due to the technology-centered nature of this problem, researchers and practitioners have been dealing with it in an implementation-based way. As pointed out by Carver and Tai (1991), the implementation of concurrent languages is threefold: it comprises a compiler, an execution environment, and an operating system (OS). By modifying some of these elements, researchers have been developing tools for deterministically executing concurrent programs. There are essentially three steps in implementing such tools (Carver and Tai, 1991):

(i) Defining the format of a synchronization sequence in terms of the synchronization constructs in the language;

(ii) Coming up with a way to transform concurrent programs so that information concerning synchronization sequences is collected during execution. Modified programs must be equivalent to originals except for the instrumentation to capture synchronization events;

(iii) Developing a way to transform programs so that they replay executions based on previously recorded synchronization sequences.

We set out to tackle this problem from an HLL VM perspective. Usually, steps two and three are carried out atop the execution environment using instrumentation libraries or some debugging facilities. We conjecture that by capitalizing on the facilities provided by modern HLL VMs it is possible to reduce the number of technologies needed to solve this problem. Apart from removing non-determinism, we intend to use the information about executions to enforce that each subsequent execution covers a different schedule (i.e., synchronization sequence) from the ones previously explored.

Next section further elaborates on the rationale behind considering HLL VMs as a reasonable basis for building an integrated software testing environment. It also describes our objectives as well as the reasons that led us to settle on using a JVM implementation.

1.2 Motivation, Objective, and Rationale

Testing tools are usually built on top of HLL VMs. As a result, they often end up tampering with the emergent computation. Usually, the additional layers between the program under test and the underlying HLL VM introduced by testing tools can have a pernicious effect on the emergent computation. Furthermore, during run time, when tools need to carry out computations about themselves (or the program under test) they must turn to costly metaprogramming operations (e.g., reflection). To examine a running program, for example, a tool has to perform introspection operations (i.e., inspecting state and structure). Likewise, to change the behavior or structure of the program under test during run time, tools have to resort to intercession (Lee and Zachary, 1995).

Apart from the overhead incurred by reflective operations, tools that implement weak mutation rely heavily on state storage and retrieval. By storing state information these tools factor out the expense of running all mutants from the beginning. Nevertheless, such computational savings are only possible at the expense of a significantly larger memory footprint (Fleyshgakker and Weiss, 1994).

The objective of this research is to investigate whether modern HLL VMs are a cost-effective technology for supporting software testing. Towards this end, we set out to extend the infrastructure provided by a full-fledged JVM with software testing support.

The rationale behind arguing that HLL VMs provide a sound basis for building an integrated mutation testing environment is that they bear a repertoire of runtime data structures suitable for accommodating the semantics of mutation testing. First, by capitalizing on existing runtime structures it is possible to decrease the amount of storage space required to implement weak mutation: from within the execution environment it is easier to determine what needs to be copied, narrowing the scope down and thus reducing storage requirements. Second, there is no need to resort to costly reflective operations since runtime information are readily available at HLL VM level. Third, by reifying mutation analysis concepts (i.e., turning them into first-class citizens within the scope of HLL VMs) it is easier to take advantage of high-end optimization and memory management features that are common in mainstream HLL VM implementations, e.g., just-in-time (JIT) compilation and garbage collection (GC). Lastly, a further advantage of building on HLL VMs data structures is that they make it possible to exert greater control over the execution of mutants.

As stated, the inherent non-determinism of concurrent programs makes software testing and debugging even more challenging. Therefore, a key missing element in modern execution environments is the ability to deterministically re-execute multithreaded programs. Since HLL VMs are the locus of control during execution, we conjecture that these execution environments contain facilities that can be extended to prune away non-deterministic behavior and enforce that subsequent executions cover distinct thread schedules from the ones previously run.

The rationale behind settling on using a JVM realization to implement our VM-based mutation analysis environment is that, apart from being by far the most used HLL VM implementation within academic circles (Durelli et al., 2010), implementations of such execution environment have sophisticated, built-in multithread support. This makes for an infrastructure more suited to our integrated software testing environment because both the fork-and-join model to speed up mutants execution and the deterministic replayer heavily rely on thread support.

1.3 Conventions Used Throughout this PhD Dissertation

Throughout this PhD dissertation, *Italic* is used for emphasis, introducing new terms, subscripts, and superscripts. `Typewriter` is used for Java operators and keywords, method and variable names, bytecodes, and URLs that appear in the text. When method names are mentioned in the main body of text, trailing parentheses are omitted. Whenever needed, a method's name is followed by a list of parameter types enclosed in parentheses, this is used to distinguish a given method from the other overloaded versions. Symbols ❶, ❷, ❸, and ❹ are used to draw the reader's attention to important information in figures and listings. Since some figures show our HLL VM being used from the command line, another convention we adopted is a symbol for the command line: throughout this document the `$` symbol (i.e., dollar sign) from *bash*, which is the default shell on many GNU/Linux distributions and on Mac OS X, is used.

1.4 Structure of this PhD Dissertation

The remainder of this document is organized as follows. Chapter 2 introduces background on virtualization and how this concept has been implemented in software. Two categories of virtual machines are described: system-level and process-level virtual machines. Particular emphasis is given to HLL VMs, which operate at process level, thus the characteristics of these implementations are described next. Two HLL VMs are discussed, namely, the P-machine and the JVM. We provide an overview of the P-machine due to its historical importance and key influence on the design of the JVM instruction set. Following the discussion of the internal organization of the JVM and its instruction set is a description of the JVM implementation we chose, namely, Maxine VM.

Chapter 3 provides background on mutation testing, its core hypotheses, and problems. Particular emphasis is given to the computational cost problem. The chapter outlines studies aimed at reducing the computational expense of mutation testing. Such an overview can be seen as an updated version of the more comprehensive survey carried out by Jia and Harman (2011), but whose scope is limited to approximation and cost reduction techniques. The chapter covers cost reduction techniques that are shown to fall into two broad categories: *(i)* techniques to reduce the number of generated mutants and *(ii)* techniques to reduce execution costs. The chapter concludes with a brief discussion of academic tools that implement some cost reduction techniques or simply automate mutation testing. Chapter 3 forms the foundation for the subsequent chapter where we present our HLL VM-based mutation system, which embodies some of the techniques discussed in this chapter.

Chapter 4 introduces our novel run-time optimization technique for speeding up mutation testing. The emphasis in this chapter is on describing how we augmented Maxine VM to support mutation testing. It is also discussed how our VM-integrated implementation further speeds up the execution of mutants by forking new threads to execute them. Next, we describe the experiment we carried out to evaluate our proof-of-concept implementation. The shortcomings of our implementation are discussed next, followed by a discussion of some related work. We end the chapter with a discussion of the benefits of our HLL VM-based mutation implementation and how it can be further improved.

In Chapter 5 we shift our attention to concurrent programs. We introduce the notion of processes and threads. This is followed by a discussion of how these abstractions are represented at language level. Next, the challenges posed by concurrency are outlined. The chapter focuses mainly on the implications that the non-determinism of concurrent programs have for testing and debugging. Some of the problems introduced by concurrency are highlighted. We then turn to a discussion of how researchers have been trying to overcome these concurrency-related problems. The techniques that have been used to detect those problems are divided into main categories: static techniques and dynamic techniques. By elucidating these techniques, this chapter motivates our implementation described in Chapter 6. The chapter also includes a discussion on how researchers have been investigating the application of mutation testing to concurrent programs.

Chapter 6 describes how we retrofitted record-and-playback and interleaving exploration capabilities into Maxine VM. Our implementation is built around Java's built-in locking mechanism. Therefore, this chapter details the way in which our record-and-playback implementation takes advantage of how the locking mechanism is realized by JVMs and Java bytecodes. Our implementation breaks down the problem of reproducing multi-threaded programs into two steps: record and replay. These steps entail code transformations, which are outlined in this chapter. Furthermore, the way our implementation uses information about previously recorded executions to explore new thread schedules is also described. An usage example is presented. Then, the results of the evaluation we carried out are highlighted. The chapter concludes with a discussion of related work as well as the shortcomings of our implementation.

Chapter 7 revisits the dissertation's research problem and RQs. It also summarizes the main contributions of this research, the current limitations of the resulting HLL VM, and suggests future research directions. Finally, the chapter concludes with a discussion of the relevance of this research and an overview of the challenges faced when extending a meta-circular HLL VM as Maxine VM.

Chapter

2

High-level Language Virtual Machines

This chapter covers background on virtual machines. It emphasizes two types of virtual machines, namely, system-level and process-level virtual machines. To elucidate the purposes of these virtual machines and the differences between them, the abstraction levels within which they operate are described. We are concerned with augmenting process-level virtual machines used to implement portable programming languages, which are known as HLL VMs, ergo special emphasis is given to this type of managed execution environments. Two such execution environments are described: the P-machine and the JVM. The P-machine and its instruction set played a pivotal role in simplifying the porting of the Pascal programming language, which led to a widespread adoption of the language. Because of its historical relevance and due to the fact that the instruction set used by such HLL VM inspired the JVM instruction set, a discussion of the P-machine is used to introduce some concepts related to HLL VMs. As a JVM was chosen to be extended in this thesis, an overview of its structure and intermediate languages is presented. In addition, the chosen JVM implementation is detailed.

The remainder of this chapter is organized as follows. Section 2.1 contextualizes how concepts such as abstraction and interfaces can be used to overcome complexity. Section 2.2 is concerned with describing how virtualization is employed to overcome the limitations posed by the use of abstraction and interfaces. Section 2.3 briefly describes system-level and process-level virtual machines. In addition, Section 2.3 focuses on the P-machine to illustrate some of the concepts related to HLL VMs. Section 2.4 outlines the main runtime data areas of the JVM as well as its intermediate instruction set. After such outline, the chapter delves into the particulars of our chosen implementation in Section 2.6. Section 2.7 makes concluding remarks.

2.1 Overcoming Complexity through Abstraction Levels and Well-defined Interfaces

A modern, general-purpose computer system is an assemblage of hardware and software subsystems that bears great inherent complexity. On the hardware side, it is commonplace to have at least one processor, storage resources, peripheral devices for performing input and output (I/O), and networking infrastructure. As for the software side, it usually encompasses an OS, several graphics and networking libraries, and application programs.[1] All in all, an intricate system. In order to circumvent such complexity and establish the collaboration and interaction among the constituent subsystems, computer systems are broken down into *levels of abstraction* separated by *well-defined interfaces* (Smith and Nair, 2005b).

Abstraction has had a ubiquitous role in several disciplines, e.g., mathematics. From a computer science viewpoint, the use of abstractions is mostly concerned with *information hiding* (Parnas, 2002). Frequently, within such context, information hiding is also referred to as *indirection.* In a sense, whether one has either direct or indirect access to a computational element has to do with the amount of detail that is hidden. According to Colburn and Shute (2007), computer scientists create levels of indirection through information hiding. Hence, they conclude that levels of indirection can be understood as levels of abstraction.

Within the context of computer systems development, orchestrating computational systems using multiple levels of abstraction allows implementation details at lower levels to be ignored, thereby simplifying the development of elements at higher levels. In general, lower levels are implemented in hardware. Since low-level elements have real properties and are made up of parts that can be physically connected, the bottom levels of a computer system are usually termed *physical levels.* By contrast, higher levels, whose elements are implemented in software and unlike physical elements are more malleable, are called *logical levels.*

The interaction between the aforementioned abstraction levels is achieved through well-defined interfaces. By hiding the intricacies of abstraction levels behind interfaces, both hardware and software designers are able to work in a more independent fashion (Smith and

[1] *Application program*, *user-level program*, *user-level application*, *user program*, and *user application* are interchangeably used throughout this document.

Nair, 2005a). An example of well-defined interface is the *instruction set architecture* (ISA), which stands at the boundary between hardware and software (Hennessy and Patterson, 2006). At this boundary, after deciding on a certain ISA, it is possible for processor designers and software engineers to work in tandem; the former developing processors that implement the chosen ISA and the latter writing compilers that translate high-level languages (HLLs) to the target ISA. As long as both groups adhere to the ISA, compiled software will execute properly on a platform[2] incorporating a processor geared towards the selected ISA (Smith and Nair, 2005a,b). Another example is the OS interface, which is seen from higher levels as a set of functions. This interface is important because it dictates the interplay between the physical and logical parts and hides most cumbersome low-level details from higher level logical elements of computer systems.

Despite the advantages of well-defined interfaces, there is a downside to relying on them: subsystems developed to a certain interface will not work with those designed for other interfaces. For example, application programs distributed as compiled binaries depend on both an ISA and an OS interface. Even OSes are tied to specific computer structural organizations, e.g., they may be tailored towards either uniprocessor or multiprocessor machines. Thus, while there are a multitude of distinct physical and logical elements that perform the same function (e.g., different I/O devices, ISAs, OSes, and HLLs), in practice, such diversity leads to reduced interoperability.

2.2 Virtualization

A workaround for the previously mentioned constraint posed by the use of heterogeneous interfaces is *virtualization*. Virtualizing a subsystem consists in mapping its interface and all visible resources onto the interface and resources of another subsystem, which is actually implementing the interface being virtualized. In other words, virtualization involves the establishment of an isomorphism that maps a *guest* subsystem to a real *host* subsystem. More formally, this isomorphism maps the guest state to the host state as shown by the function V in Figure 2.1. Similarly, the set of operations, e, that modifies the guest state (e changes the guest state S_i to S_j) is mirrored by a corresponding sequence of operations, e', in the host that performs an equivalent state modification (changes S'_i to S'_j).

Although abstraction also conforms to the aforementioned definition for virtualization, Smith and Nair (2005b) distinguish the two concepts. According to them, abstraction

[2]The terms *machine* and *platform* are interchangeably used throughout this document. However, according to Smith and Nair (2005a), platform is currently the term more in vogue.

differs from virtualization in that the latter does not hide details, thus the complexity presented by the host is typically the same as that in the virtualized guest.

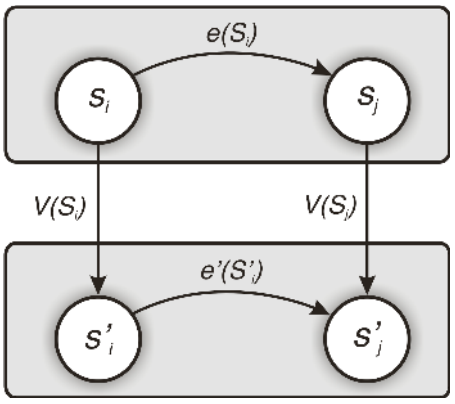


Figure 2.1: Virtualization. Formally, it consists in establishing an isomorphism between a guest and a host. This figure is adapted from Smith and Nair (2005b).

As emphasized by Smith and Nair (2005b), through virtualization the host subsystem can also be transformed so that it appears to be several virtual subsystems. Taking a hard disk as an example, virtualization could be applied in order to partition it into many smaller virtual disks. In such scenario, each virtual disk would be mapped to the real hard disk by being implemented as a large file on the real disk, as illustrated in Figure 2.2. In addition, the files representing virtual disks would play a supportive role to the virtualizing software managing the mapping between virtual disk contents and real disk contexts (the function V in the isomorphism shown in Figure 2.1). In this way, each virtual disk would appear to have a number of logical tracks and sectors and writing to one of the virtual disks (the function e in the isomorphism) would entail both a file write and a real disk write in the host system (the function e' in the isomorphism).

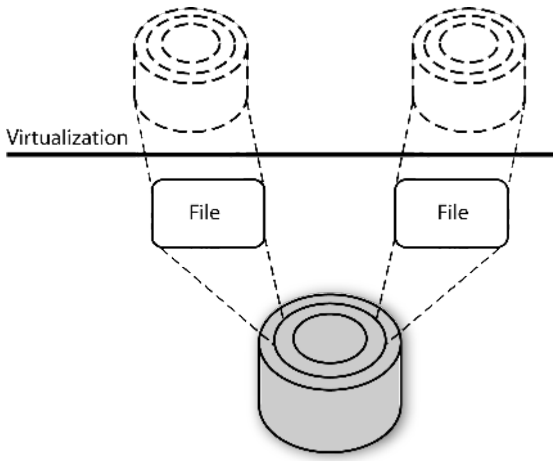


Figure 2.2: Implementing virtual disks through virtualization. This figure is adapted from Smith and Nair (2005b).

2.3 Virtual Machines

Software implementations of the concept illustrated in Figure 2.1 have been around for over four decades. Such software layers are termed *virtual machines* (VMs). The first implementations took the concept of virtualization one step further by virtualizing entire machines instead of subsystems. Such software replicas of real machines were mainly tailored towards multiplexing low-level hardware resources, thereby maximizing the use of such virtualized physical properties. However, over the years, VMs have been proven useful for coping with a number of other computing problems. As a result, this technology has spanned a wide range of domains. Furthermore, due to the versatility of this technology, currently, there is a broad diversity of VMs and they have been designed, built, and investigated not just by OSes and hardware developers but by language and compiler designers as well (Rosenblum, 2004; Treese, 2005; Killalea, 2008). Naturally, each sort of implementation has its particular characteristics and goals. Therefore, before going on to describe the types of VMs, it is worth outlining the boundaries in which these technologies are employed. Next section describes these boundaries by providing an overview of the structure of computer systems.

2.3.1 Computer-system Architecture

To elucidate what VMs are, an examination of what is meant by machine and platform is in order. In either case, the meaning is a matter of perspective. From the perspective of a process running a user program, the machine – or platform – consists of the following elements: logical memory address space, user-level registers, and instructions that allow the execution of the underlying process. Usually, the running process has no direct access to I/O operations. Instead, these operations are accessible only through the OS, which entails that the process has to perform system calls. Consequently, the machine, from the perspective of a process, is a combination of both the OS and the underlying hardware. As can be seen in Figure 2.3(a) the interface via which processes communicate with the underlying machine is called *application binary interface* (ABI).

In contrast, from the perspective of an OS, a whole system is implemented by the bare hardware. Such a system is an execution environment able to simultaneously support a multitude of processes. Within this execution environment (which persists over time) processes (which are usually transient in nature) share resources and a file system, both of which are managed by the OS. Therefore, from the perspective of a system, the machine is implemented by the underlying hardware alone. Figure 2.3(b) depicts this system-level perspective within which the platform is implemented by the hardware and whose interface is the underlying ISA.

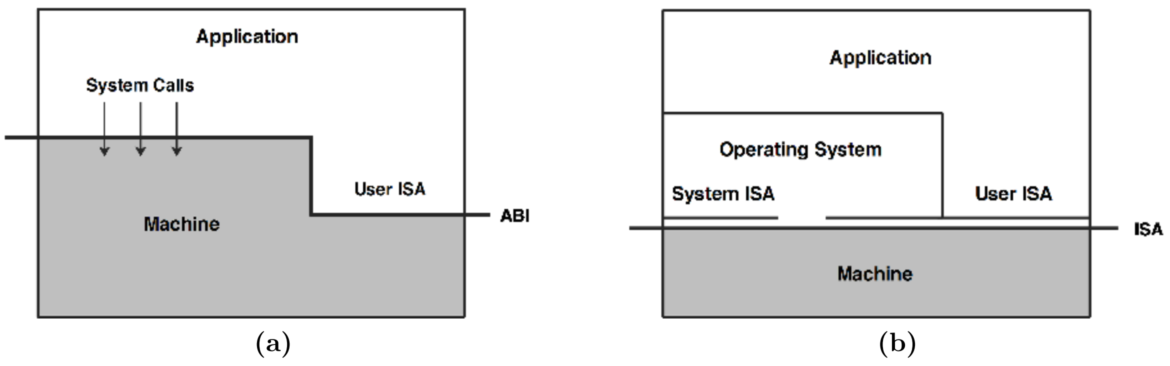


Figure 2.3: The underlying platform from two perspectives, namely, application and OS. From the perspective of an application, the underlying platform is a combination of OS and hardware. The communication between an application and the underlying platform occurs through the ABI, as shown in (a). From the perspective of the OS, the platform is the hardware itself. OSes and platforms interact mostly through the ISA, as shown in (b).

Due to the fact that most VM implementations are closely tied to the boundaries shown in Figures 2.3(a) and (b), Smith and Nair (2005a,b) propose a candidate taxonomy that characterizes VM according to two major categories: process and system VMs. Process VMs run atop the OS/hardware combination and, as the name of the category suggests, are aimed at supporting an individual process as long as it runs. The scope tackled by process VMs is shown in Figure 2.3a. In contrast with process VMs, the scope dealt by system VMs is wider, encompassing support for an OS and its user-level applications. Figure 2.3b illustrates the ISA boundary at which system VMs stand.

Next section presents the rationale for VMs designed to virtualize entire machines.

2.3.2 System VMs

The earliest system-level VMs date back to the late 1960s and early 1970s (Creasy, 1981; Smith and Nair, 2005a). At that time, computational resources were quite expensive and therefore scarce. Another factor that hindered users from making the most of computational resources was the presence of either mono-task or simple batch OSes. Consequently, system-level virtualization was essentially conceived to pursue better usage of expensive mainframe hardware. International Business Machines (IBM) pioneered the efforts toward developing the first system VM implementations. The first system-level VM was the IBM System/360 Model 40 VM (Smith and Nair, 2005a), but one of the best known implementations was IBM's VM/370 (Goldberg, 1974; Seawright and MacKinnon, 1979; Tanenbaum, 2007).

Whole-system VMs can be described as facsimiles of a real existing machine (Popek and Goldberg, 1974). They act as a duplicate of the underlying real machine aimed at allowing multiple guest OS environments to simultaneously run on a single host hardware platform. In essence, each of these guest OSes has the impression of being running alone on raw hardware (Rosenblum, 2004). Similarly to the example shown in Figure 2.2, system VMs replicate a single set of hardware resources so that it appears to be multiple independent sets of hardware resources. Thus, through virtualization, system VMs foster better hardware usage by preventing it from either becoming idle or being underused.

Within a system-level VM environment, the layer of software responsible for keeping track of the real resources of the host platform is called *VM monitor* (VMM). VMM layers are also known as *hardware-level VMMs* (Rosenblum, 2004), *type 1 hypervisors* (Tanenbaum, 2007), and *non-hosted hypervisors* (Carbone et al., 2008). The classic organization, followed by the first implementations, places the VMM layer on bare hardware and the guest system VMs atop of it (Goldberg, 1974). In such an organization, it is through the VMM layer that the host platform resources are made available to the VMs. Since only the VMM layer runs in privileged mode, whenever one of the guest OS performs an operation that demands interaction with the underlying hardware, the VMM *(i)* intercepts the operation, *(ii)* checks its correctness, and *(iii)* performs it. Moreover, it is up to the VMM to decide how the resources will be accessed by the VMs, e.g., resources may be either partitioned or time-shared (Smith and Nair, 2005b).

Another hallmark of VMMs is that some of the resources made available by them may not have corresponding physical counterparts in the platform being virtualized. In such cases, VMMs may emulate the desired resources by combining software and other resources that are available on the underlying platform (Smith and Nair, 2005a). Examples of this type of VM include Xen (Barham et al., 2003), VMware ESX (Waldspurger, 2002), and Microsoft Hyper-V (Kelbley and Sterling, 2008).

Apart from type 1 hypervisors, there is another type of system-level VM, namely, *type 2 hypervisor*. Differently from type 1 hypervisors that interact directly with the underlying hardware, type 2 hypervisors interface with OSes. These system-level VMs run as user-level applications on top of the host OS. Thus, they are also known as *hosted VMs*. Examples of type 2 hypervisors include Oracle VM VirtualBox[3](Oracle Corporation, 2011), VMware Server(VMware Inc., 2011), Parallels Workstation (Parallels Holdings Ltd., 2011), and Microsoft Virtual PC (Microsoft, 2011).

[3]This hosted VM was formerly known as Sun xVM VirtualBox.

It is worth mentioning that most of the early platforms were not able to support system VMs properly. The ones that were able to host such VMs had to resort to inadequate techniques due to their unsuitable architectures. Currently, however, most platforms are capable of supporting VMs. Virtualization at the system level has progressed a lot, and currently this technology is a boon to developers. According to Spinellis (2012), properly taking advantage of virtualization allows for better development environments, easier distribution, more efficient testing, and improved infrastructure management.

2.3.3 Process VMs

Typically, programs are compiled down to executable binaries that conform to a specific ABI. That is to say, these programs include features of a certain ISA and an OS, as shown in Figure 2.3(a). Given that such features are hardcoded into the executable binaries, an important restriction is that a program compiled to a certain platform will not run on a distinct platform (Smith and Nair, 2005b). One application of process-level VMs is to overcome this limitation. Process VMs virtualize an individual program in order to port it to a platform other than the one it was originally designed for. The virtualization occurs at the ABI level (Smith and Nair, 2005a,b). As shown in the right side of Figure 2.4, when an application is being virtualized by a process VM, it does not directly interact with the host platform. Instead, every operation performed by the guest application is emulated by the process VM.

The main drawback of this approach is that it is only effective on a case-by-case basis (Smith and Nair, 2005b). For example, if one wanted to run IA-32 binaries on different platforms such as PowerPC and MIPS, two process VMs would have to be developed.

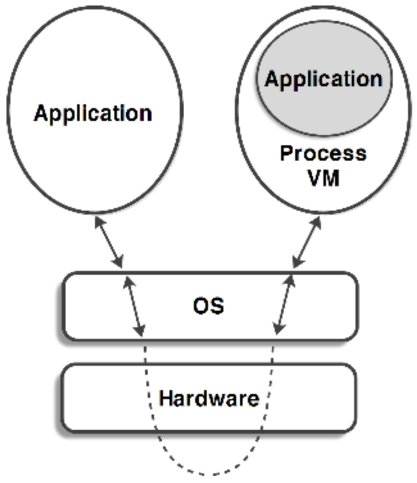


Figure 2.4: An overview of how applications are virtualized by process VMs. This figure is adapted from Smith and Nair (2005b).

Instead of treating portability as an after-the-fact consideration, a more effective way of achieving cross-platform portability is devising an abstract machine that minimizes all platform-specific characteristics. By compiling applications to the language implemented by the abstract machine the portability problem boils down to having an implementation of the abstract machine in each target platform. This is the approach taken by language designers to create programs in a platform-neutral format. Implementations of such abstract machines are referred to as HLL VMs. This sort of process VM is discussed in the next sections.

2.3.4 HLL VMs

As described in preceding sections, compiled application programs are tied to a particular OS and ISA. As a consequence, before the concept of HLL VMs was around, porting user-level applications to a different platform involved at least recompilation (Smith and Nair, 2005b). However, before getting around to tackling the porting of application programs, a porting of the underlying compiler had to be made available for the target platform, which was a major source of problems and demanded considerable effort.

Porting a HLL compiler to another platform involved facing several technical hurdles as *(i)* the complexity of mapping from one conventional ISA to another and *(ii)* addressing the quirks of the target ISA. In practical terms, a great deal of the complexity related to porting a compiler stems from bridging the semantic gap between an HLL and the target platform ISA. Thus, aimed at easing the porting of compilers, developers started adopting an additional step to compilation: rendering HLLs to intermediate representations geared towards abstract machines, and then translating such intermediate languages to machine code (Groves and Rogers, 1980). In such context, the term abstract machine is used to refer to intermediate target languages and related architectures that act as an intermediate stage to compile HLLs (Diehl, 1998). Next subsections present further information on both related concepts.

By adding an intermediate compilation step, porting compilers to a new platform mostly entailed changing their back-ends to transform instructions of the intermediate language into machine code of the target platform. Subsequently, developers realized that the porting process could be sped up even further by implementing an interpreter to emulate the intermediate language on the target platform rather than compiling down to machine code. By eschewing the implementation of a machine-code-generating compiler, it was possible to get a full-fledged execution environment up and running quickly. Such approach proved to be very effective due to the simplicity of the interpreters that were needed (Klint, 1981; Ballarin et al., 1988).

Gough (2001, 2005) argues that the main benefit of using an abstract machine was compiler factorization. By compiling to an intermediate language the problem of porting N different languages to M differente platforms was reduced to implementing N front-ends and either M back-ends or M interpreters rather than $N \times M$ (Mogensen, 2011). In addition to improving portability, intermediate languages can be designed from the ground up to take into account important features of their respective HLL. This results in efficient implementations of the underlying HLL (Smith and Nair, 2005b).

Differently from system VMs whose role is to be a facsimile of a real machine, HLL VMs have been conceived as abstract machines with no physical counterpart. One of the main goals of these process VMs is to provide an execution environment for specific HLLs, whereas the motivation behind other process VMs is to emulate conventional ISAs.

Next subsection gives further detail on HLL VMs by describing a landmark in technical development of HLLs: the Pascal P-machine, the HLL VM that pioneered the implementation of the foregoing concepts.

2.3.4.1 Pascal P-machine

The use of an abstract machine and its virtual ISA to define program representation for compilers dates back to the mid-to-late 1970s when the early implementations of the Pascal programming language were developed (Wirth, 1996; Mickel, 2000; Gough, 2001; Smith and Nair, 2005b; Gough, 2005; Scott, 2009; Wilhelm and Seidl, 2011). According to Wirth (1996), the first Pascal implementations were a threefold technology: *(i)* a compiler and *(ii)* an abstract machine and *(iii)* its virtual ISA, which were respectively known as P-compiler, P-machine, and P-code (Nelson, 1979; Gabbrielli and Martini, 2010). Such technology was realized by a toolkit known as P-kit (Scott, 2009), which was made available as the following:

- a Pascal compiler written in Pascal that generated P-code;
- the same compiler already translated into P-code;
- a P-code interpreter written in Pascal.

Aimed at getting Pascal up and running on a certain platform, developers had to translate the P-code interpreter into some locally available language. This was a straightforward task since the required interpreter was small. After porting the interpreter, the P-code version of the compiler could then be run and used to render arbitrary Pascal programs into P-code, which could in turn be run on the ported interpreter. This solution to portability issues played a pivotal role in driving Pascal's widespread adoption. More importantly, it established such concept as an alternative for implementing platform-neutral HLL (Gregg et al., 2005). Circa late 1978 there were over 80 Pascal implementations on distinct hosts and by 1980 manufactures of workstations adopted Pascal for system programming (Wirth, 1996). Due to its ease of porting, Pascal thrived in academic settings (Scott, 2009). Up to mid 1990s, Pascal used to be the most popular choice for introductory computer science courses (Thalmann and Thalmann, 1978; Brilliant and Wiseman, 1996; Pears et al., 2007; Goosen, 2008).

P-machine set a standard for later HLL VM implementations. A number of HLL VM developers have gleaned knowledge from the P-machine experience, insofar as the designs of P-machine and their HLL VMs overlap. Thus, a variety of contemporary HLL VMs are akin to P-machine. Basically, the main differences between the currently most widely used HLL VMs and P-machine are that the latter did not allow for *(i)* network support, *(ii)* automatic memory management, and *(iii)* security features (Smith and Nair, 2005b).

To some extend, the success of Pascal can be ascribed to P-code. Since Pascal P-machine is stack-based, P-code includes stack-manipulation instructions (e.g., instructions for pushing onto and popping from the stack). This characteristic made it easy to translate from P-code to any conventional ISA because no constraint was posed on the minimum amount of registers in the target platform. Furthermore, virtual ISAs can convey more semantic information than machine code (Aycock, 2003), and using a stack-based virtual ISA yields even smaller program representations. Low-memory-footprint programs were of paramount importance at that time given that many platforms had small hard drives and supported only small amounts of main memory (Smith and Nair, 2005b).

Using a virtual ISA to specify programs also has its shortcomings though. It is up to the HLL VM to bridge the gap between the virtual ISA and the target platform ISA. As mentioned, the simplest approach to implement an execution environment was through a straightforward interpretation. However, the main disadvantage of an interpretive approach is its slow execution speed. Since then, researchers have been devising a variety of techniques for improving HLL VM execution engines (i.e., the component responsible for emulating the virtual ISA) to cope with performance woes. In the next section we elaborate on the techniques that have been used to implement execution engines. By doing so, we also further categorize the different representations emulated by and related to HLL VMs.

2.3.4.2 HLL VM Execution Engines

There are various sorts of programs whose only purpose is to serve as an execution engine for other application programs. Traditionally, within the context of HLL VMs, execution engines are realized through a plain interpreter or a runtime compiler (Craig, 2005). In either case, a spectrum of techniques can be used to implement them (Klint, 1981; Kazi et al., 2000; Davis and Waldron, 2003). Some of the modern HLL VMs even use an amalgam of interpretation and compilation (Prokopski and Verbrugge, 2008; Wilhelm and Seidl, 2011).

Execution engines are heavily influenced by the abstraction level of the representation that they emulate. Rau (1978) categorizes program representations into three broad classes:

High-level representation (HLR): HLLs such as Ruby (Flanagan and Matsumoto, 2008; Thomas et al., 2009), Scala (Wampler and Payne, 2009; Odersky et al., 2011), Java (Sabharwal, 1998; Niemeyer and Knudsen, 2005), Perl (Schwartz et al., 2011), and Python (Lutz, 2009) fit into this class. As Arnold et al. (2005b) point out, HLRs convey semantic information at a high level of abstraction and aim at facilitating human understanding rather than machine execution. Usually, HLRs comprise a myriad of operators, control structures, problem-oriented constructions, and so-called syntactic sugar. These elements do not affect their expressiveness but provide alternative ways of coding that can be clearer or more succinct. It is worth emphasizing that an HLR may not be an HLL. For instance, although the programming language C (Kernighan and Ritchie, 1988; Ritchie, 1996) is an HLR, it may be argued that it is not an HLL per se.

Directly interpretable representation (DIR): an intermediate representation whose syntax is fairly simple in comparison to an HLR. Usually, DIRs contain a small set of elementary operators and, differently from conventional ISAs, these intermediate representations are designed to be free of quirks[4] and implementation-specific constraints (Smith and Nair, 2005b). P-code is an example of DIR. Currently, DIRs are metadata-rich representations, encompassing both data and instructions. A mainstream example that reflects the current bias towards metadata-centered DIRs is Java bytecodes (Gosling, 1995; Engel, 1999; Lindholm and Yellin, 1999)[5].

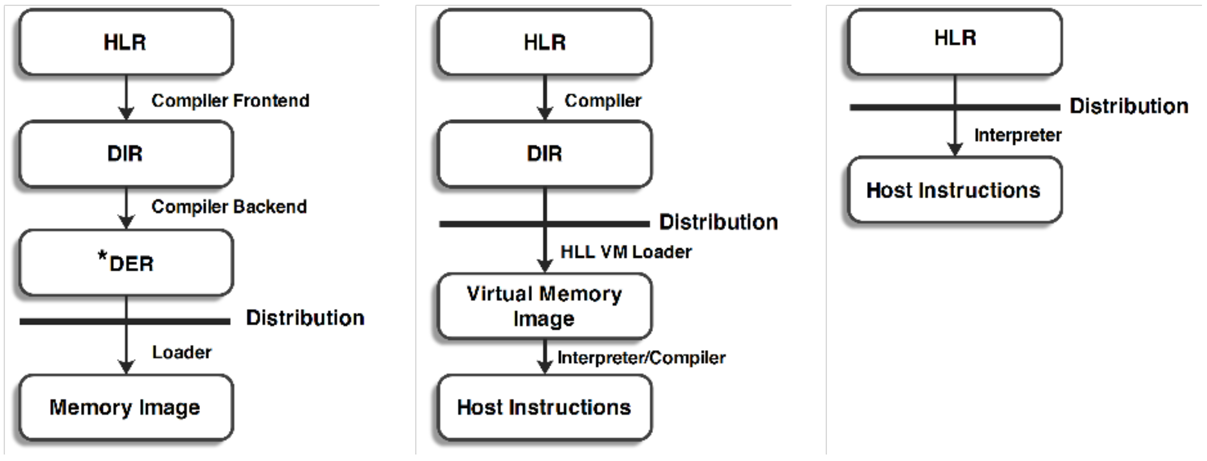
Directly executable representation (DER): an executable binary representation tied to the host platform. Usually, DERs stand at the lowest abstraction level.

Approaching HLL VMs from the viewpoint of the aforementioned taxonomy, it is possible to further refine their definition as follows: abstract machines tailored towards bridging the semantic gap between either of the first two classes (i.e., HLR and DIR) and the underlying host platform's ISA (i.e., a DER representation). Within such context,

[4]Usually, ISAs have special features that have to be addressed in a way that deviates from what would be expected from them, such non-conventional features are referred to as *"quirks"* (Smith and Nair, 2005b).

[5] Another prominent example of metadata-centered intermediate language is Microsoft's Common Intermediate Language (CLI). However, CLI does not fit well into the classification proposed by Rau (1978) because it is not intended to be interpreted (Smith and Nair, 2005b).

execution engines play a central role: they are responsible for the emulation process through which upper-level representations are translated into DERs. The way execution engines carry out such translation differs from static (i.e., ahead-of-time) compilation, which is the fashion in which traditional compilers operate (Klint, 1981; Aho et al., 2007; Wilhelm and Seidl, 2011). Figure 2.5 contrasts the conventional steps in getting from an HLR to a DER both using static compilation and HLL VM execution engines.



(a) (b) (c)

Figure 2.5: Conventional steps in getting from HLR to a DER using static compilation (a), an HLL VM that uses an DIR (b), and an HLL VM that inspects its HLR in a character-by-character fashion (c).

Figure 2.5(a) shows the steps taken by a static compiler. At first, the compiler frontend renders the HLR into a DIR. Then the backend takes the previously created DIR and compiles it down to a DER, which is either a form of assembly code or a sort of relocatable object code (Elsworth, 1979; Aho et al., 2007). In effect, object code cannot be considered a DER because it lacks symbolic information that is resolved at load time. As depicted in Figure 2.5(a), at run time, a loader embodies missing symbolic information into object code representations, turning them into full-blown DERs.

An HLL VM implementation may use either of the following translation approaches: *(i)* interpretation of the DIR and *(ii)* direct interpretation of the HLR. In the former case, Figure 2.5(b), a static compiler is employed to render the underlying HLR into a DIR. After being translated to the respective DIR, programs are ready for being distributed as well as executed on a compatible HLL VM implementation. At run time a loader is invoked aimed at checking whether DIRs are in compliance with the HLL VM specification and turning them into runtime data structures that are dependent on the HLL VM implementation. After such step, the execution engine manages the emulation process by interacting with runtime data structures. An example of language whose translation process involves statically compiling the HLR to a DIR is Java (Diehl, 1998; Lindholm and Yellin, 1999; Niemeyer and Knudsen, 2005).

As illustrated in Figure 2.5(c), there are also HLL VMs implementations whose execution engines inspect HLRs in a character-by-character fashion, performing actions accordingly (Klint, 1981). As a result, these execution engines rely on repeated lexical and syntactic analysis of the HLR, which often incurs in severe performance degradation. Two examples of languages that are distributed in their HLR form are Ruby and Python.

2.4 A Primer on the JVM and Its Key Runtime Data Areas and Virtual ISA

This section describes the structure of the JVM according to its specification (Lindholm and Yellin, 1999; Lindholm et al., 2012). We describe the JVM as an abstract computing machine in terms of its relevant runtime data areas and virtual ISA. Accordingly, implementation details are deliberately omitted in this section. More details on the JVM implementation we chose will be given in Section 2.6.

The JVM specification states that the JVM is the cornerstone of the Java programming language (Li, 1998; Lindholm and Yellin, 1999; Lindholm et al., 2012). In general, the additional abstraction layer provided by a JVM implementation stands between Java programs and the underlying OS and platform. Technically, this layer is the main responsible for the Java technology's cross-platform portability and for emulating the virtual ISA.

Early prototype implementations of the JVM, developed at Sun Microsystems, Inc., were specifically targeted to a hand-held device.[6] Currently, since the JVM specification does not mandate any particular host OS or host platform (Radhakrishnan et al., 2001), there are implementations that emulate the JVM on a wide range of platforms and devices (Gough and Corney, 2000; Lindholm et al., 2012) as embedded systems (Levis and Culler, 2002; Koshy and Pandey, 2005; Simon and Cifuentes, 2005; Simon et al., 2006; Aslam et al., 2008; Brandner et al., 2009; Thomm et al., 2010), smart-card devices (Guthery, 1997; Oestreicher, 1999; Azevedo et al., 2005), desktops (Haase and Guy, 2007), and servers (Downey, 2007), to name a few. Furthermore, apart from the conventional pure software implementations, the JVM design allow for a variety of implementation choices ranging from pure hardware (Hardin, 2001; Berekovic et al., 1997; O'Connor and Tremblay, 1997; McGhan and O'Connor, 1998; Tan et al., 2006; Puffitsch and Schoeberl, 2007; Schoeberl, 2012) as well as a combination of hardware and software (i.e., hardware and software co-design implementations) (Fong et al., 2012).

[6]The device in question was called $*7$; otherwise known as "Star7" or "StarSeven" (Haines and Potts, 2003, pages 7–8).

The JVM was conceived with the purpose of supporting only the Java programming language.[7] Nevertheless, this technology was designed in such a way that it knows nothing of the Java programming language per se. In effect, JVM implementations understand only a particular binary format, i.e., the class file format (Lindholm et al., 2012). A class file contains instructions (i.e., bytecodes) as well as a host of static metadata. Such a feature makes it possible to port other languages to the JVM. In fact, the JVM has successfully become the premier multi-language execution environment. So far, a host of languages has been successfully ported to the JVM (Gough and Corney, 2000; Allman, 2004). Prominent examples are the scripting languages (Ousterhout, 1998) Ruby and Python whose Java implementations are called JRuby (Edelson and Liu, 2008; Nutter et al., 2011) and Jython (Pedroni and Rappin, 2002; Allman, 2004), respectively. Another scripting language that is dynamically compiled to the platform-neutral format employed by the JVM is Groovy (Koenig et al., 2007). Apart from these scripting languages, multi-paradigm programming languages, such as Scala (Odersky et al., 2011), have also been exploring the applicability of the JVM as a target execution environment.

According to its specification (Lindholm and Yellin, 1999; Lindholm et al., 2012), the JVM uses several runtime data areas to execute programs. Some runtime data areas are created upon JVM start-up and destroyed when the JVM exits, whereas others are per thread data areas, and as such, they are created along with a thread and destroyed as soon as their respective thread finishes execution. The main runtime data structures are outlined in the next subsections.

2.4.1 Heap

The global storage area for holding objects and arrays is managed as a *heap*. The heap is created on JVM start-up so that several other data storage structures are allocated from it. Objects allocated on the heap are not explicitly deallocated: despite the fact that there is an instruction that allocates memory for new objects, there is no instruction for explicitly deallocating memory. Rather, heap storage is managed by an automatic storage management (i.e., GC) system. The heap can be either of a fixed size or dynamically expanded and contracted. The specification also underlines that the memory for the heap does not need to be contiguous (Lindholm et al., 2012). One runtime data structure that is allocated from the heap is the *method area*, which is detailed in the following subsection.

[7]From its inception until 1994, the Java programming language was called Oak (Waldo, 2010, pages viii–ix).

2.4.2 Method Area

Each instance of the JVM has a *method area* that is shared among all threads. This runtime data area is created on virtual machine start-up and stores per-type structures such as the runtime constant pool (Section 2.4.3), field and method data, as well as code for methods and constructors, including the special methods related to class, interface, and instance initialization (Lindholm and Yellin, 1999). Although the JVM specification does not mandate the location of the method area or policies used to manage compiled code, usually, this structure is allocated in the heap. Similarly to the heap, the method area can be either of a fixed size or dynamically expanded and contracted.

2.4.3 Runtime Constant Pool

A *runtime constant pool* is a per-class or per-interface runtime representation of the `constant_pool` table in a Java class file. When a class or interface is loaded into the JVM, a runtime constant pool is allocated in the method area. Such a per-type runtime data structure comprises several sorts of constants that range from numeric literals to method and field symbolic references (Lindholm and Yellin, 1999); the former are known at compile time and the latter are resolved at run time. Although a runtime constant pool contains a wider range of data than a typical symbol table of a conventional programming language, both runtime representations perform a similar function.

2.4.4 Stack

In addition to the global (e.g., heap) and per-type structures (e.g., runtime constant pool), JVM also contains per-thread and local runtime data structures. These drive the execution of Java programs. Many threads can be spawned at run time. When a new thread is created, it is associated with a JVM *stack* (Lindholm and Yellin, 1999), which manages method invocation and returns by storing local variables and partial results. Basically, JVM stacks use two local structures to hold data: a *program counter* (PC) register and a *frame*. PC registers keep track of the code being executed by the underlying thread. The PC register points into the method area, indicating the current instruction (Engel, 1999). If the current method is not native, the PC register holds the address of the method instruction currently being executed. Frames are more complex than PC registers. Next subsection describes frames.

2.4.5 Frame

As mentioned, frames are one of the runtime data structures allocated from the JVM stack. Frames play a fundamental role in the following: storing parameters and temporary data, performing dynamic linking, returning values for methods, and dispatching exceptions (Kazi et al., 2000; Lindholm et al., 2012). These runtime data areas are fundamental to method invocation: a new frame is created whenever a method is invoked. Thus, each frame corresponds to a method invocation that has not returned yet. The top frame of a stack is called the *active frame* (Engel, 1999) or *current frame* (Lindholm et al., 2012).

When control returns from a method, whether it is a normal or an uncaught exception, the active frame is popped off the underlying JVM stack. Upon normal method completion, the result of the method's computation, if any, is transferred from the active frame to the invoker. Moreover, the frame below the active frame becomes the new active frame. Frames use an *array of local variables* and an *operand stack* to pass parameters and store intermediate results.

The parameters being passed to the invoked method are stored in its array of local variables. The array's length is determined at compile time. The array takes two consecutive local variables to hold the value of a `long` or `double` type, other stack types (`int`, `float`, and reference) take up only one local variable.[8] Each entry on the local variable array is accessed by indexing. The initial index of the parameters varies depending on the type of the method. If the method is a class method, its parameters begin at the first element of the local variable array, that is, at index zero. Instance methods, have the first element of their local variable arrays reserved for the this pseudovariable, thus their parameters begin at index one (Engel, 1999; Lindholm et al., 2012).

All computations performed in the context of methods take place in the operand stack. The operand stack is empty upon frame creation, and values are pushed onto or popped from the stack during execution. The operand stack acts as a last-in-first-out stack (Lindholm et al., 2012). The depth of a stack fluctuates during execution. Nevertheless, similarly to the local variable array, the maximum depth of the operand stack is determined at compile time (Engel, 1999; Craig, 2005) and `long` and `double` types take up two operand stack slots. In addition, it also stores parameters to be passed to methods as well as method results. Figure 2.6 gives an overview of how stacks, frames, arrays of local variables, and operand stacks are organized.

Besides the internal organization, another important element of an HLL VM is its ISA. Usually, this sort of ISA is designed with portability in mind, thereby keeping hardware-specific characteristics to a minimum.

[8]The types `boolean`, `byte`, `short`, and `char` are treated as `int`.

In the next section we give details about the JVM's virtual ISA.

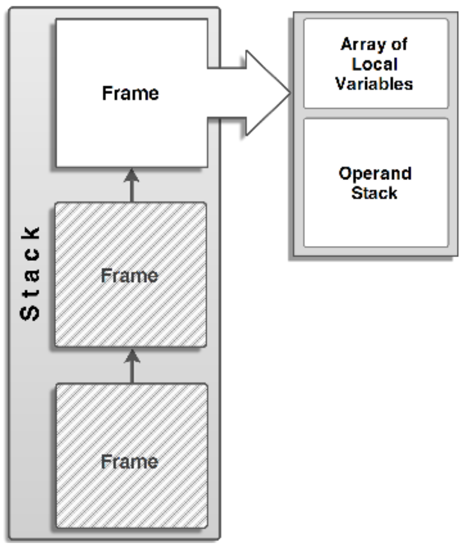


Figure 2.6: An overview of the organization of JVM stacks and their frames. As can be seen, a stack is composed of frames. A frame, in turn, includes the state of one Java method invocation. Whenever a thread calls a method, a new frame is pushed onto that thread's stack. Once the method returns, the frame is popped from the stack. In white, the frame on top corresponds to the current method (i.e., the last method invoked by the current thread). Frames in gray represent methods that are currently inactive (i.e., methods waiting for control to return from the current method).

2.4.6 The JVM Instruction Set: Java Bytecodes

As pointed out earlier, the JVM knows nothing about the Java programming language. Instead, programs for this HLL VM are expressed in terms of an intermediate representation called bytecodes (Engel, 1999; Lindholm et al., 2012). This virtual ISA is akin to the ISA of most computer architectures. However, differently from the ISAs employed by computer architectures, which are often register-based ISAs, bytecodes are a stack-based virtual ISA (Gosling, 1995; Lindholm et al., 2012).

According to Gosling (1995), the design of Java bytecodes drew inspiration from Pascal P-code. Similarly to P-code, a great deal of bytecode instructions are typed. In other words, primitive types have specific bytecode instructions to operate on them. Bytecodes follow a naming convention in which the first letter indicates what type the instruction operates on (Engel, 1999). For instance, the `iadd` instruction adds the top two values on the operand stack and replaces them with the sum, provided that both values are of type `int` (arithmetic instructions are further described later on in this section). Analogously, the mnemonic of the instruction that performs the same operation on `long` variables is `ladd`. In these mnemonics the letters `i` and `l` indicate that the instructions operate on `int` and `long` variables, respectively. The mnemonic naming convention is outlined in Table 2.1.[9]

Table 2.1: Mnemonic type letters. This table was adapted from Engel (1999).

Mnemonics	
Letter	**Type**
`a`	`reference`
`b`	`byte` or `boolean`
`c`	`char`
`d`	`double`
`f`	`float`
`i`	`int`
`l`	`long`
`s`	`short`

Basically, a bytecode comprises a one-byte opcode field (hence the name), which represents the operation to be performed, and zero or more operands (Craig, 2005). Using one byte to represent opcodes limits the maximum size of the virtual ISA to 256 instructions. As of this writing, the JVM ISA consists of 212 instructions, the remaining 44 opcodes are reserved for future extensions. Describing all 212 instructions is out of the scope of this document. To limit the scope of our discussion, we will break the JVM virtual ISA down into subsets of instructions, presenting examples to illustrate some characteristics of this virtual ISA. Most instructions fall into one of the following subsets (Craig, 2005):

- Control-flow instructions;
- Data-manipulation instructions;
- Stack-manipulation instructions.

Structured programming constructs are not supported at bytecode level (Engel, 1999). Instead, control-flow instructions are provided to transfer control from one location to another location within a method (e.g., branch instructions). A family of conditional branch instructions compares the topmost stack element with zero: the JVM pops the top stack element and performs the test on it (Craig, 2005; Lindholm et al., 2012). The value on the top of the stack must be an `int`. The mnemonics for opcodes of this family follow the naming convention `if<cond>`, where `<cond>` represents the comparison to be performed. Different

[9]It is worth emphasizing that the prefix letters used in descriptor fields are different from the ones used in bytecode mnemonics. Descriptor fields use uppercase letters and in some cases they do not correspond to the lowercase letters in Table 2.1. For instance, the prefix letter for `boolean` types in descriptor fields is `Z` rather than `b`. More information about description fields can be found in the JVM specification (Lindholm and Yellin, 1999; Lindholm et al., 2012).

comparisons with zero are performed depending on the instruction. Table 2.2 gives an overview of the `if<cond>` family. Additionally, the instructions in this family include two bytes that represent the offset to the next instruction (Figure 2.7). If the comparison of the top element against zero succeeds, the JVM calculates the offset based on the instruction's last two bytes (i.e., `branchbyte1` and `branchbyte2`) and branches.[10] This causes the JVM to continue execution with the instruction indicated by the offset (Lindholm et al., 2012).

Table 2.2: Instructions in the `if<cond>` family.

`if<cond>` **Family**	
Mnemonic	**Description**
`ifeq`	succeeds if and only if the top of the stack is equal to zero
`ifne`	succeeds if and only if the top of the stack is not zero
`iflt`	succeeds if and only if the top of the stack is less than zero
`ifle`	succeeds if and only if the top of the stack is less than or equal to zero
`ifgt`	succeeds if and only if the top of the stack is greater than zero
`ifge`	succeeds if and only if the top of the stack is greater than or equal to zero

`if<cond>`
`branchbyte1`
`branchbyte2`

Figure 2.7: Format of the `if<cond>` instruction family.

The JVM ISA also has unconditional branches. An example of unconditional branch instruction is `goto`. Similarly to instructions in the `if<cond>` family, the `goto` instruction also includes two unsigned bytes that are used to calculate the offset (Figure 2.8). After executing a `goto` instruction, the JVM branches to the instruction specified by the second and third operands (e.g., `branchbyte1` and `branchbyte2`).

`goto`
`branchbyte1`
`branchbyte2`

Figure 2.8: Format of the `goto` instruction.

As comparing variables to `null` is a frequent operation in Java programs, there are two special instructions that check whether a given reference variable is `null`: `ifnull` and `ifnonnull` (Engel, 1999). An in-depth description of the other control-flow bytecodes is provided in the JVM specification (Lindholm and Yellin, 1999; Lindholm et al., 2012).

Most instructions in the data-manipulation subset are concerned with arithmetic operations. There are bytecode instructions for performing addition, subtraction, multiplication, division, remainder, and negation operations. These instructions have the following

[10]Transferring control from one instruction to another is called *branching* (Engel, 1999).

mnemonics: `<x>add`, `<x>sub`, `<x>mul`, `<x>div`, `<x>rem`, and `<x>neg`, respectively. As these are type-specific instructions, `<x>` represents the type upon which they operate. For instance, `isub` performs subtraction operations on `int` variables and `ddiv` divides its two `double` operands. The forms that arithmetic instructions can take are listed in the first part of Table 2.3.

Table 2.3: Data-manipulation instructions.

Opcode	byte	short	int	long	float	double	char	reference
Arithmetic Operations								
<x>add			iadd	ladd	fadd	dadd		
<x>sub			isub	lsub	fsub	dsub		
<x>mul			imul	lmul	fmul	dmul		
<x>div			idiv	ldiv	fdiv	ddiv		
<x>rem			irem	lrem	frem	drem		
<x>neg			ineg	lneg	fneg	dneg		
Logical Operations								
<x>and			iand	land				
<x>or			ior	lor				
<x>xor			ixor	lxor				
Type-conversion Operations								
i2<x>	i2b	i2s		i2l	i2f	i2d	i2c	
l2<x>				l2i	l2f	l2d		
f2<x>			f2i	f2l		f2d		
d2<x>			d2i	d2l	d2f			

Note that there is no `boolean` type in Table 2.3. The reason is that the JVM uses `int` variables to represent types that are less than 32-bits in size (e.g., `char`, `byte`, and `short`). Therefore, most operations on `boolean` values are carried out by instructions that operate on `int` or `long` values (Lindholm et al., 2012). Examples of operations that support only `int` and `long` values are the logical operations: `<x>and`, `<x>or`, and `<x>xor`. The forms that these instructions can take appear in the second part of Table 2.3.

Due to the fact that some operations apply only to integers (e.g., logical operations), the JVM virtual ISA also includes type conversion instructions. The mnemonics of these type-changing instructions follow the form `<x>2<y>`, where both `<x>` and `<y>` are letters in Table 2.1: `<x>` represents the type that the instruction converts from and `<y>` is the type that the instruction converts to (Engel, 1999). So, for instance, `i2b` converts the `int` value on the top of the stack into a `byte` value. Type-conversion instructions are outlined in the third part of Table 2.3.

In stack-based ISAs, the operands must be pushed onto the underlying stack before any computation takes place. Being a stack-based ISA, the JVM virtual ISA includes instructions to move data from the main storage and local storage (i.e., heap and array of local variables) to the operand stack. Some of the instructions that push and pop local variables onto the stack have the following mnemonics: `<x>load` and `<x>store`, where `<x>` is a letter in Table 2.1 and indicates the type upon which these instructions operate (Lindholm et al., 2012). The `<x>load` instruction family loads variables from the array of local variables to the operand stack of the current method. As shown in Figure 2.9(a), these instructions are followed by an index into the array of local variables (local variables are accessed by indexing, as explained in Subsection 2.4.5). For example, `fload 1` pushes the first local variable of an instance method onto the operand stack. The `<x>store` instruction family, shown in Figure 2.9(b), performs the opposite operation: it pops the value on the top of the stack and stores it into the local variable indicated by the index operand. For example, `istore 2` moves the value on the top of the stack to the `int` local variable at position 2. As pointed out by Engel (1999), for efficiency purposes, there are special instructions that load and store the variables at indexes 0, 1, 2, and 3. These special instructions incorporate the index in their mnemonic. They follow the format `<x>load<n>`, where `<x>` is a letter in Table 2.1, indicating the type upon which the instruction operate, and `<n>` is any number from 0 to 3. For instance, `fload_2` has essentially the same effect as `fload 2`.

<x>load
index
(a)
<x>store
index
(b)

Figure 2.9: Formats of the `<x>load` and `<x>store` instruction families.

The JVM virtual ISA is not limited to the aforementioned subsets. Some characteristics of the Java language are reflected in bytecodes. To cite an instance, there are instructions that support object orientation. The instruction `new`, for example, instantiates a new object. Storage for the new instance is allocated from the heap, and a reference for it is placed on the stack.

There are two concurrency-related instructions in the JVM ISA: `monitorenter` and `monitorexit` (Lindholm and Yellin, 1999; Marr et al., 2009; Lindholm et al., 2012). These two instructions implement the Java built-in synchronization mechanism, which is detailed in Section 6.1.

2.4.6.1 Advantages of Java Bytecodes

Much of the success of the JVM can be ascribed to its virtual ISA. In a sense, bytecodes are an improvement over P-code because of the following twists: there is more type and data information, restrictions are imposed on the use of the stack, and there is reliance on symbolic references (Gosling, 1995). As Smith and Nair (2005b) remark, the degree of platform independence achieved by modern virtual ISAs exceeds the one once provided by P-code. The main reason is that instead of focusing only on the ISA aspects of portability, contemporary virtual ISAs also emphasize how to encode data in a platform-neutral fashion.

Given that operands are implicitly fetched from the operand stack, rather than being explicitly represented along with instructions as register numbers, programs compiled to bytecodes stay compact (Shi et al., 2008; Lindholm et al., 2012). Another advantage of being stack based is that it greatly simplifies the implementation of JVMs on register-poor architectures.

As reported by Shi et al. (2008), stack-based ISAs lead to more compact programs, mainly when compared to register-based ISAs. This compactness makes stack-based ISAs suitable for network computing environments because a dense program representation saves network bandwidth.

2.5 JVM Implementations

The JVM is described as an abstract machine in its specification. As such, some implementation details are not part of this specification. For example, implementors are free to choose any GC algorithm and internal optimization. According to Lindholm et al. (2012), specifying all these technical details would constrain the creativity of implementors. Furthermore, an over-specified JVM definition would compromise the implementation of specification-compliant JVMs for certain devices. For example, demanding that every JVM implementation includes a JIT compiler would make it difficult to design a JVM implementation for memory-constrained devices. For these reasons, there are many different implementations of the JVM, some are implemented in C or C++ and others are implemented mostly in Java itself. A non-exhaustive list of JVM implementations is shown in Table 2.4.

An example of JVM tailored to resource constrained devices is Squawk (Simon and Cifuentes, 2005; Simon et al., 2006). Squawk is written in Java and runs on the bare metal, avoiding the need for an OS (Simon et al., 2006). In addition, Squawk is an interpreted JVM, which also reduces its memory footprint. Other JVMs designed to small devices are the following: NanoVM (Till Harbaum, 2013), FijiVM (Pizlo et al., 2010), KESO (Wawersich et al., 2007; Thomm et al., 2010), Darjeeling (Brouwers et al., 2008), TakaTuka (Aslam et al., 2008), Jelatine (Agosta et al., 2006), and Maté (Levis and Culler, 2002).

As for desktop implementations, Jikes RVM (Research Virtual Machine) was the first JVM implemented in Java (Jikes RVM Project, 2013). It started out as an internal research project at IBM, but since 2001 Jikes RVM has been available as an open source project. Currently, Jikes RVM runs on many platforms. It has become a research project whose goal is to provide a versatile implementation that can be used to prototype HLL VM technologies. Thus, many researchers have adopted Jikes RVM as their research infrastructure to explore new adaptive optimization and memory management techniques (Durelli et al., 2010). Other widely used desktop implementations are HotSpot (Oracle Corporation, 2013c) and Maxine VM (Oracle Corporation, 2013a; Wimmer et al., 2013). We chose to extend Maxine VM, thus a description of this HLL VM is provided in the next section.

Table 2.4: A non-exhaustive list of JVM implementations. The implementations are listed in alphabetical order.

		Execution Engine		
Name	**Creator**	**Interpreter**	**JIT**	**Availability**
CACAO	Vienna University of Technology	No	Yes	Free
Darjeeling	Niels Brouwers	No	No‡	Free
FijiVM	Fiji Systems Inc.	No	No‡	Free
Guaraná	Spin-off from Kaffe	Yes	Yes	Free
HotSpot†	Sun Microsystems, Oracle	Yes	Yes	Free
J9	IBM	Yes	Yes	Free
JRockit	Oracle	No	Yes	Free
JamVM	Robert Lougher	Yes	Yes	Free
JamaicaVM	Aicas GmbH	?	Yes	Commercial
Jelatine	Gabriele Svelto	Yes	No	Free
Jikes RVM	IBM	No	Yes	Free
Joeq	John Whaley	Yes	Yes	Free
KESO	Erlangen-Nürnberg University	No	No‡	Free
Kaffe	Transvirtual Technologies	Yes	Yes	Free
LaTTe	Seoul University, MASS Lab	No	Yes	Free
Maté	University of California, Berkeley	Yes	No	Free
Maxine VM	Sun Microsystems, Oracle	No	Yes	Free
NanoVM	Till Harbaum	Yes	No	Free
SableVM	Sable Research Group	Yes	No	Free
Squawk	Sun Microsystems, Oracle	Yes	No	Free
Steamloom	Spin-off from Jikes	No	Yes	Free
TakaTuka	University of Freiburg	Yes	No	Free

†JDK Edition.
‡Ahead-of-time compilation.

2.6 Maxine VM

Maxine VM is an open-source JVM from Oracle Laboratories (formerly Sun Microsystems Laboratories). It is intended to be used as a replacement of the Java HotSpot VM (Oracle

Corporation, 2013c), which is shipped with the Java Development Kit (JDK) from Oracle and the OpenJDK (Oracle Corporation, 2013b). It is a meta-circular HLL VM (i.e., written in the same language that it realizes). In contrast with conventional HLL VMs that are implemented in a low or intermediate level language such as C, Maxine VM is mostly written in Java. Only a small part of Maxime VM, called the *substrate*, is written in C. Figure 2.10 contrasts the design of conventional JVMs with the design of a meta-circular JVM.

In the Java part, two aspects leverage the meta-circularity in Maxine VM's design: it *(i)* integrates with Oracle's standard Java Development Kit (JDK) packages, implementing their down-calls in Java, and *(ii)* relies on a fast baseline compiler and an optimizing compiler that translates the HLL VM itself.

The optimizing compiler generates a boot image of Maxine VM in an "ahead-of-time" fashion (Oracle Corporation, 2013a). The boot image is not a native executable but a binary blob targeted at the platform for which the image was generated. The boot image contains data required by the HLL VM until it can begin loading further classes and compiling methods on its own (e.g., a pre-populated heap and a binary image of method compilations). However, since the boot image is not executable by itself, bootstrapping Maxine VM entails invoking the substrate (Wimmer et al., 2012), which loads the boot image and then transfers control of the execution to the HLL VM.

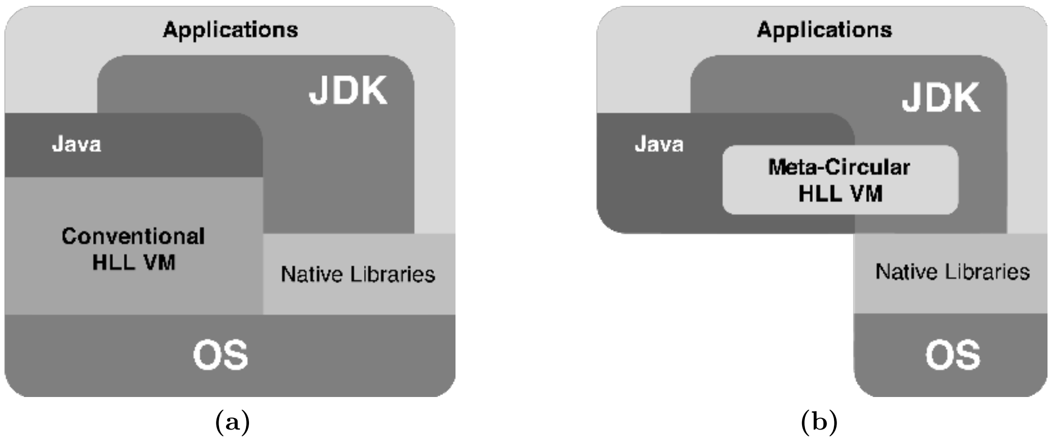


Figure 2.10: A conventional HLL VM design (a) and a meta-circular HLL VM design (b). This figure is adapted from Mathiske (2008).

The main reason Maxine VM was chosen is because its design was tailored to support HLL VM research. Its modular design allows for replacing entities with different implementations, making it a flexible testbed for trying out new ideas and prototyping HLL VM technology. For example, it is possible to plug in alternate implementations of subsystems such as GC and compilation.

Maxine VM is also a compelling platform because of its high performance. This JVM implementation does not use an interpreter; instead it uses a JIT-only strategy. In effect, a tiered compilation strategy takes place during run time. Methods are compiled with a lightweight non-optimizing JIT compiler (i.e., baseline compiler), which in a single forward pass translates Java bytecodes into pre-assembled native instructions. Afterwards, the JIT optimizing compiler may be triggered to further optimize methods that are frequently invoked (*hotspots*). Some of the optimizations performed by the JIT optimizing compiler are method inlining and whole-method register allocation (Bebenita et al., 2010).

Another advantage of Maxine VM is that it has a debugger and object browser, called Inspector (Würthinger et al., 2010). This co-designed companion tool allows the perusal of runtime information at multiple levels of abstraction. For example, objects can be inspected either abstractly or in terms of the platform-dependent memory layout. Likewise, methods can be viewed as source code, bytecodes, or native instructions.

As of this writing, Maxine VMs uses a simple semi-space copying collector as its default GC. Due to the fact that the current GC performs poorly, several other GCs are under development: a basic flat mark-sweep GC with allocation based on segregated list without compaction, a region-based mark-sweep GC with linear allocation per-region, and a generation GC with a copying nursery (Oracle Corporation, 2013a).

2.7 Concluding Remarks

Computer systems are complex structures encompassing many layers of interacting components in both software and hardware. Within this context, virtualization plays the role of interconnection technology. This chapter summarized the underpinning concepts of virtualization. As discussed, the concept of virtualization is implemented by VMs. Over the years, VMs have pervaded a wide range of domains. The versatility of this technology has led to a vast diversity of VMs. To put the most representative examples of this technology in perspective, we presented them according to the taxonomy proposed by Smith and Nair (2005a). Their taxonomy divides VMs into two major types: *(i)* system VMs and *(ii)* process VMs. The boundaries in which these technologies are employed were highlighted. Given the focus of the research herein described, in the present chapter a type of process VM was emphasized, namely, HLL VMs.

Two HLL VMs were outlined: the Pascal P-machine and the JVM. P-machine was described because of its historical importance: it popularized the concept of using an intermediate representation to enhance portability. As for the JVM, its intrinsic advantages over statically compiled binaries have led to a widespread adoption on various platforms ranging from web servers to low-end embedded systems. Many studies, including this one, have used this HLL VM (Durelli et al., 2010). This study, more specifically, uses a JVM implementation named Maxine VM, whose main characteristics were described in this chapter.

In the next chapter, we give an introduction to mutation testing. We focus on describing techniques aimed at speeding up mutation testing. Thus, the following chapter is intended to lay the theoretical foundation required for understanding our approach to weak mutation testing and our proof-of-concept implementation, which are discussed in Chapter 4.

Chapter 3

Mutation Testing

Mutation testing has been around for more than forty years. Although it has been proven effective as a fault revealing technique, its main downside is that it is computationally expensive. Aimed at dealing with this computational cost problem, researchers have striven to create approximation and cost reduction techniques. As discussed throughout this chapter, these techniques need to strike a balance between reducing computational cost and maintaining fault revealing effectiveness. It is not only the cost of running mutants that has to be taken into account, thus researchers have also been tackling the problem of reducing the amount of generated mutants. The approximation and cost reduction techniques outlined in this chapter set the foundation for the next chapter in which we describe our HLL VM-based cost reduction technique.

This chapter is not intended to provide a comprehensive survey of all existing work in the area. Instead, emphasis is given to approximation and cost reduction techniques. The chapter is organized as follows. Section 3.1 introduces mutation testing, its core hypotheses, and problems. Section 3.2 describes how mutation is used from a procedural standpoint. Section 3.3 presents an overview of studies that aim to mitigate the computational cost associated with carrying out mutation testing. This section can be seen as an updated version of the survey carried out by Jia and Harman (2011), but whose scope is limited to approximation and cost reduction techniques. Section 3.3 briefly describes some academic mutation tools. Section 3.4 outlines efforts to apply mutation testing to concurrent programs. Concluding remarks are given in Section 3.5.

3.1 The Theory of Mutation Testing

The first contributions to mutation testing can be traced back to the 70's: a student paper by Lipton (1971) and the paper by Hamlet (1977). Nevertheless, the paper by DeMillo et al. (1978) is the one that is usually cited as the seminal reference. Since then, research into mutation testing has come a long way (Offutt and Untch, 2001; Jia and Harman, 2011).

Mutation testing is effective at helping to identify test data that is adequate to uncover real faults (Andrews et al., 2005; Do and Rothermel, 2006; Jia and Harman, 2011). However, since that mimicking all potential faults for a given program is very costly, mutation testing focus on a subset of these faults. Faulty versions that are close to the correct version of the program under test are emphasized, thereby reducing the number of alternate versions of the program that needs to be generated (Morell, 1988). In doing so, it is expected that uncovering how the faulty versions deviate from the original program is enough to emulate all faults. This idea is based on two hypotheses: the *competent programmer hypothesis* and the *coupling effect* (DeMillo et al., 1978).

The competent programmer hypothesis was first proposed in the seminal work of DeMillo et al. (1978). According to such a hypothesis, programmers are competent, thereby they tend to write programs that are close to being correct. Thus, it can be assumed that when a program written by a competent programmer has faults, those faults are quite simple; they are slight syntactic deviations from the correct program. As a result, when applying mutation testing only straightforward syntactical changes are performed. That is, only the faults that mimic mistakes that a competent programmer would make.

DeMillo et al. (1978) also proposed the coupling effect. The coupling effect has to do with the characteristics of the faults used in mutation analysis. It states that test data that is sensitive enough to distinguish a correct program from versions containing simple faults is also able to uncover more complex faults. Offutt (1989, 1992) expanded on this by proposing two hypotheses, the *coupling effect* and the *mutation coupling effect*, and providing a definition of simple and complex faults. According to his definition, a simple fault is a single syntactical change made to a program, whereas a complex fault is made up of several changes. As stated by Offutt (1992), the coupling effect hypothesis boils down to complex faults being coupled to simple faults so that a test data that is capable of uncovering simple faults will also end up exposing most of the complex faults. As for the mutation coupling effect hypothesis, complex mutants are coupled to simple ones in such a way that a test data set that detects all simple mutants will also detect most of the complex mutants (Offutt, 1992). Aside from the research by Offutt (1989, 1992), several studies have been conducted in order to experimentally support the coupling effect hypothesis (Morell, 1983; Wah, 1995, 2000, 2003; Kapoor, 2006).

Before going into detail about studies on mutation testing, in the next section we describe how mutation is used from a procedural standpoint. Then, a number of studies tailored towards turning mutation testing into a more practical techniques are discussed in Section 3.3.

3.2 An Overview of the Mutation Testing Process

As described in the previous section, mutation testing is centered around the idea of devising test data for uncovering the seeded faults. Within such context, these faults are slight syntactic changes made to a given program. The elements that describe such syntactic changes are called *mutation operators*[1] (Ammann and Offutt, 2008; Ahmed et al., 2010; Hu et al., 2011). Also, as mentioned, a hallmark of these changes is that they are analogous to mistakes that programmers make. Typical mutation operators are applied to source code and modify expressions by replacing, inserting, or deleting either operators or variables. When mutation operators are applied to the program under test, they result in faulty versions of the program called *mutants* (Offutt et al., 1996a). As an illustration, Listing 3.1 shows a Java method with examples of mutation operators. The sample method in Listing 3.1 contains four mutated lines (❶, ❷, ❸, and ❹), each line represents a mutant generated from the original method. Mutants ❶ and ❷ replace the relational operator `<` with `>` and `>=`, respectively, ❸ replaces one variable reference with another, and mutant ❹ inserts the post-increment operator `++`.

Before starting analyzing the generated mutants, test data is designed to execute the original program and assert its correctness. To do so, the original program (i.e., `P`) must be executed against a test set `T`. Whenever `P` is incorrect, it needs to be fixed before starting the mutation analysis. After successfully running the original program, the generated mutants (`M`) are run against `T` with the goal of fulfilling the following conditions (Ammann and Offutt, 2008):

- **Reachability:** The mutated location in the mutant program must be executed.
- **Infection:** The state of the mutant program must be incorrect after the faulty location is executed.
- **Propagation:** The infected state must be propagated to some part of the output.

[1]In the mutation testing literature, mutation operators are also known as *mutagenic operators*, *mutant operators*, *mutagens*, and *mutation rules* (Offutt and Untch, 2001).

Listing 3.1: Method `min` and three mutants: mutants ❶ and ❷ replace the relational operator `<` with `>` and `>=`, respectively. So mutants ❶ and ❷ are examples of mutants generated from the mutation operator called relational operator replacement (ROR). Mutant ❸ replaces the variable `d2` with `d1`, which is an instance of the application of the mutation operator called scalar variable replacement (SVR). Mutant ❹ inserts the post-increment operator `++`, which is an example of the sort of syntactical changes made by the arithmetic operator insertion shortcut (AOIS). This listing was adapted from Offutt et al. (1996a) and Ammann and Offutt (2008).

```
public static double min(double d1, double d2) {
  if (d1 < d2) return d1;
   ➥ ❶ if (d1 >  d2) return d1;
      ❷ if (d1 >= d2) return d1;
  else return d2;
   ➥ ❸ else return d1;
   ➥ ❹ else return d2++;
}
```

This is called the *RIP* model. When a test case $t \in$ `T` satisfies the RIP model by causing the output of the mutant to be different from the output of the original program, the test case is said to *kill* the mutant; otherwise, the mutant is said to be *alive*. Afterwards, new tests are run against all live mutants, killed mutants (also called dead mutants) no longer need to remain in the mutation testing process. Dead mutants can be removed from the process because the faults represented by these mutants have already been detected by `T`. Therefore, they have fulfilled the goal of identifying an effective test case. However, if a mutant has the same output as the original program for all possible test cases, it is said to be *equivalent*. Despite the fact that equivalent mutants differ from the original program due to the syntactical changes they have gone through, they are functionally equivalent to the original program (Jia and Harman, 2011). In fact, the syntactical change in an equivalent mutant cannot be considered a fault but an optimization or de-optimization of the underlying code (Usaola and Mateo, 2010). For example, mutant ❹ in Listing 3.1 is an example of equivalent mutant because it will always return the same result as the original program.

As the process carries on, test cases are added to `T` and run against live mutants until either all mutants are killed or the tester decides that `T` is good enough. Eventually, the goal of mutation analysis is to obtain a *mutation score* of 100%. The mutation score is the percentage of nonequivalent mutants that have been killed (Ammann and Offutt, 2008; Naik and Tripathy, 2008), so a score of 100% indicates that `T` is able to detect all the faults represented by the mutants (i.e., `M`). To sum up, the purpose of the technique is twofold: *(i)* providing an test-adequacy criterion and *(ii)* detecting faults in the program under test.

Therefore, after applying mutation testing and achieving a mutation score of 100%, one obtains two results: *(i)* a satisfactory set of test cases (i.e., `T`) and *(ii)* a reliable original program, given that the set of test cases found no faults in `P` (Usaola and Mateo, 2010). In most cases, achieving a mutation score of 100% is impractical, so a threshold value can be established; representing the minimum value for the mutation score. Figure 3.1 gives an overview of the mutation testing process.

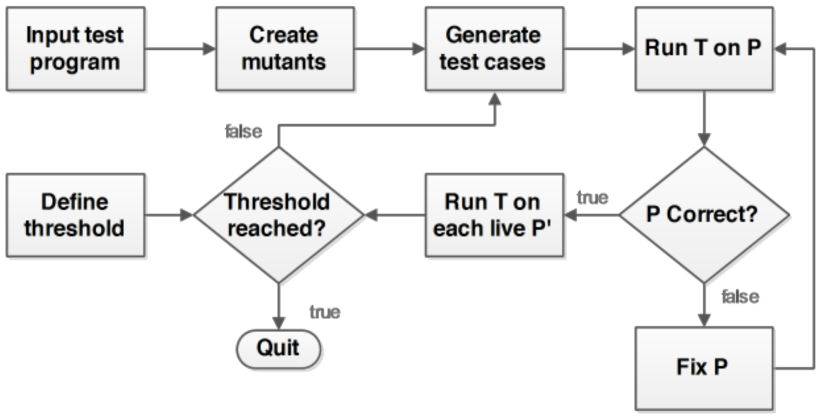


Figure 3.1: An overview of the mutation testing process. This figure is adapted from two sources: Offutt and Untch (2001) and Ammann and Offutt (2008).

From a research viewpoint, mutation testing is a mature technique (Usaola and Mateo, 2010; Jia and Harman, 2011; Offutt, 2011). Mutation testing has been used as a "gold standard" for experimentally evaluating other software testing techniques (Andrews et al., 2005; Do and Rothermel, 2006; Namin and Kakarla, 2011). However, several hurdles associated with carrying it out have been hindering its adoption as a practical testing technique. First, carrying out mutation testing also entails a lot of human effort that might make it cost-prohibitive. To be more specific, the two steps that involve human effort are the *(i) human oracle problem* (Weyuker, 1982) and *(ii)* the problem of identifying equivalent mutants (Budd and Angluin, 1982). Second, mutation testing is computationally expensive because of the high cost of running a large number of mutants against a test set.

The human oracle problem has to do with the tester having to check the program's output for each test case in order to evaluate whether the program is properly functioning for those particular inputs. However, as noted by Jia and Harman (2011), this issue is not unique to mutation testing. In fact, in many sorts of testing, checking the outcome of the test set is still required. As for identifying equivalent mutants, it is a well-known undecidable problem as proved by Budd and Angluin (1982). Therefore, the identification of equivalent mutants is often carried out manually and is labour intensive. The labor-intensiveness associated with detecting equivalent mutants is discussed by Frankl et al. (1997). Grün et al. (2009) report that it takes approximately 15 minutes to manually evaluate whether a mutant is equivalent to the original program. While it is not possible to come up with comprehensive solutions for these two hindrances to mutation testing, the literature is rife with studies exploring how to circumvent them (Jia and Harman, 2011).

Similarly to many testing techniques, mutation testing also requires tools to automate some steps of the process (e.g., mutants generation and managing the execution of the generated mutants). However, this automated tool support demands high computational cost. For instance, carrying out mutation testing, even when taking into account moderate-sized programs, results in hundreds of mutants (Usaola and Mateo, 2010). Reducing either the number of generated mutants or the execution cost has been the goal of a number of research ideas including ours, which is described in Chapter 4. Thus, in order to lay the foundation for Chapter 4, the next sections cover mutation testing cost reduction techniques by providing some background and giving an overview of what has already been investigated in this area.

3.3 Cost Reduction Techniques

Mutation testing is not a scalable criterion. As Harrold (2000) remarks, although it has been shown that mutation testing is an effective criterion, researchers have yet to come up with ways to carry out the testing efficiently. Since mutation testing was first proposed, much attention has been concentrated on reducing the computational expense of generating, compiling, and executing the mutant programs against the test set in question (Usaola and Mateo, 2010; Jia and Harman, 2011). As pointed out by the survey of Offutt and Untch (2001), approaches to overcome the computational cost of mutation testing are based on one of three strategies: *"do fewer"*, *"do smarter"*, or *"do faster"*.

Approaches that follow the "do fewer" strategy try to devise ways of reducing the mutants that need to be taken into account without incurring significant loss in terms of quality. The "do smarter" approaches are concerned with distributing the computational expense over several computers or splitting the expense up into several executions by storing run-time information between runs. The "do faster" approaches emphasize ways of boosting up the generation and mainly the execution of mutants. In this document, we followed the classification proposed in the survey of Jia and Harman (2011), which contains only two categories: *(i)* reduction of the generated mutants (which is equivalent to the "do fewer" strategy) and *(ii)* reduction of the execution cost (which combines two strategies, namely, "do smarter" and "do faster"). According to Jia and Harman (2011), the cost reduction techniques that have been most studied are selective mutation and weak mutation. These two and other cost reduction techniques are discussed in detail in the next subsections.

3.3.1 Reduction of the Number of Generated Mutants

In the light of the computational cost that stems from having to compile and run a large number of mutants against a test set, a research thrust in mutation testing has been exploring how to reduce the number of generated mutants as a way of expediting execution. Typically, mutation systems implement lots of mutation operators. As noted by Offutt and Untch (2001), the rationale was "to include as much testing as possible by defining as many mutants as possible". For instance, Mothra mutation system for Fortran (DeMillo et al., 1988; King and Offutt, 1991) supports 22 mutation operators. Due to high computational cost associated with executing the profusion of mutants generated by the mutation operators, reducing the number of generated mutants has become an important research problem.

3.3.1.1 Selective Mutation

The amount of mutants generated for a given program is roughly proportional to the product of the number of data references times the number of data objects (Offutt et al., 1996a). Some mutation operators yield far more mutants than others since they can mutate nearly every statement in the original program (Usaola and Mateo, 2010). Nevertheless, some of these mutants may turn out to be redundant. For example, two of the 22 Mothra mutation operators generate up to 40% of all mutants (King and Offutt, 1991), namely, array reference for scalar variable replacement (ASR) and SVR (see Listing 3.1 for an example of the application of SVR). In an effort to effectively reduce the number of redundant mutants, Wong et al. (1994) suggested the idea of *constrained mutation*, which consists in applying mutation testing using only the most sound mutation operators. Due to the large number of redundant mutants generated from ASR and SVR, Wong et al. suggested leaving them out of the mutation process.

Offutt et al. expanded on this idea by proposing an approximation technique called *selective mutation*, which entails selecting only mutants that are truly different from other mutants (Offutt et al., 1993, 1996a) and play a part in strengthening the test set in question. That is, selective mutation deals with ascertaining the smallest set of mutation operators that generates the best subset of mutants, without compromising effectiveness. The idea of Wong et al. was implemented by Offutt et al. (1993) as "2-selective mutation". Apart from omitting two mutation operators, Offutt et al. also experimentally evaluated omitting four ("4-selective mutation") and six ("6-selective mutation") mutation operators. Their results indicate that 2-selective mutation achieves a mean mutation score of 99.99% and a 24% reduction in the number of generated mutants. As for 4-selective mutation, it achieved a mean score of 99.84% and the number of generated mutants had a reduction of 41%. The figures for 6-selective mutation are a mean mutation score of 88.71% with a reduction of 60% in the number of mutants.

Wong and Mathur (1995) expanded on their previous work and came up with another type of reduction strategy in which mutation operators are selected based on their effectiveness. Wong and Mathur suggested using two mutation operators: absolute value insertion (ABS) and relational operator replacement (ROR) (see Listing 3.1 for an example of the application of ROR). According to them, the motivation behind choosing ABS is that test cases that kill mutants generated from this mutation operator exercise different facets of the input domain. ROR was chosen because test cases tailored to ROR mutants are sensitive to logical expressions within predicates. Their results indicate that these two mutation operators achieve a reduction of 80% in the number of mutants with a decrease of only 5% in the mutation score.

Employing a reduction strategy similar to the one proposed by Wong and Mathur, Offutt et al. (1996a) further extended their 6-selective mutation. They divided Mothra's mutation operators into three groups: *(i)* mutation operators that change statements, *(ii)* operators that modify operands, and *(iii)* mutation operators that operate on expressions. After grouping the mutation operators, they went on to omit operators from each class in turn. Their findings suggest that five operators from groups *(ii)* and *(iii)* are the key mutation operators: ABS, ROR, arithmetic operator replacement (AOR), logical connector replacement (LCR), and unary operator insertion (UOI). It turns out that according to Offutt et al., using this set of five mutation operators, almost the same coverage as non-selective mutation can be achieved.

Mresa and Bottaci (1999) devised a slightly different reduction strategy. Besides aiming for the smallest possible set of mutation operators that does not compromise the fault-finding effectiveness of the test set, they suggest that the cost of detecting equivalent mutants also needs to be taken into account. In their study, they report that it is possible to reduce the number of equivalent mutants and yet retain a reasonable degree of effectiveness.

Drawing from previous works, Barbosa et al. (2001) developed guidelines on how to select an effective set of mutation operators from a given mutation system. Such a guideline was applied to Proteum (Program Testing Using Mutants), which is a mutation system that implements a comprehensive set of mutation operators for C programs (Maldonado et al., 2001). Proteum's set of mutation operators is made up of 77 operators, of which 10 were selected after applying the underlying guidelines. A mean mutation score of 99.6% with a 65.02% reduction in the number of mutants was achieved using the selected mutation operator set.

Namin et al. address the problem of mutant selection by formulating it as a statistical problem: the problem of variable selection[2] (Namin and Andrews, 2006, 2007; Namin et al., 2008). Feature selection is the process of analyzing an entire set to find the optimal subset that will yield the best results (Kohavi and John, 1997; Blum and Langley, 1997). Applying their linear statistical approaches, they reached a subset of 28 mutation operators (out of 108 C mutation operators). According to them, their results predict the effectiveness of a test suite and reduce the number of generated mutants in 92%. Therefore, compared with the other approaches, Namin et al.'s approach achieved the highest rate of mutant reduction.

Zhang et al. (2010) compared the aforementioned reduction strategies with two random techniques. In fact, in their study, Zhang et al. examined three reduction strategies: *(i)* the reduction strategy of Offutt et al. that yielded five mutation operators, *(ii)* Barbosa et al.'s 10 mutation operators, and *(iii)* Namin et al.'s 28 mutation operators. These three reduction strategies were compared against two random mutant-selection techniques. The first random technique, given a number n, selects n mutants randomly. The second random technique comprises two stages. In the first stage, a mutation operator is chosen randomly. During the second stage, mutants generated from the chosen mutation operator are randomly selected until n mutants have been selected. Only mutants that have not been previously selected are chosen. Their experimental results suggest that the three reduction strategies are no better than picking mutants randomly, none of three reduction strategies outperforms random mutant selection in terms of effectiveness. In addition, their study indicates that random mutant selection is able to achieve competitive results even when selecting fewer mutants than each of the three reduction strategies.

Li et al. (2009) found that although selective mutation generates far more test requirements than the edge-pair, all-uses, and prime path criteria, it needs fewer tests. An important implication of this finding is that many mutants are redundant. This led researchers to postulate that mutation testing can still be quite effective with fewer mutants. Untch (2009) set out to investigate this by using a simple, direct approach: a single mutation operator. Untch used the statement deletion operator (SDL), which removes statements from the program in the hopes of impelling the tester to design tests that cause the absence of every statement to have an effect on the outcome. Despite the fact that the SDL operator does not mimic faults that a programmer would make, Untch obtained promising results. Recently, Deng et al. (2013) also investigated how the SDL operator can reduce the overlap among mutants. Deng et al. implemented the SDL operator in the muJava (Ma et al., 2005) mutation system. Then, they performed an empirical evaluation

[2]In statistics and machine learning, the problem of variable selection is also known as feature selection, feature reduction, attribute selection, and variable subset selection (Blum and Langley, 1997).

with 40 subject programs. They killed all SDL mutants for these subjects, yielding an SDL-adequate test suite. Such SDL-adequate tests were run against all muJava's mutants, achieving on average mutation score of 92% while generating 80% fewer mutants. As pointed out by Deng et al., this is a huge savings at the expense of a slight loss in effectiveness.

3.3.1.2 Mutant Sampling

Mutant sampling concerns randomly selecting a subset of mutants from the whole set (Jia and Harman, 2011). Acree (1980) and Budd (1980) pioneered research into mutant sampling. In this approach, after mutant generation, $x\%$ of the mutants are selected randomly, and the others are discarded. The focus has been on the choice of the random selection rate (e.g., $x\%$) (Jia and Harman, 2011).

Wong (1993) carried out an experiment using a random selection rate $x\%$ that ranged from 10% to 40% in increments of 5%. He found that by using only 10% of the mutants there is a decrease of 16% in terms of the mutation score that can be achieved in comparison with the entire set of mutants. This result implies that mutant sampling is effective with a $x\%$ higher than 10%. This is in line with the findings of DeMillo et al. (1988) and King and Offutt (1991).

Rather than trying to fine-tune the sample rate using increments of a-priori fixed size, Sahinoglu and Spafford (1990) devised a sampling approach that is based on the Bayesian sequential probability ratio test (SPRT) (Wald, 1945). Their approach consists in randomly selecting mutants until a statistically appropriate sample has been reached. The results reveal that their approach is more sensitive than the ones based on plain random selection due to the fact that their approach self-adjusts to the test set.

3.3.1.3 Higher Order Mutation

Higher order mutation is a relatively new form of mutation testing introduced by Jia and Harman (2009). As mentioned in Section 3.1, mutants created through traditional mutation testing differ from the program under test in only one way: they contain a single fault. According to Jia and Harman (2009), often these faulty versions are quite trivial, and as such they end up being killed easily. In contrast, mutants created through higher order mutation contain two or more faults. The fundamental idea is to generate mutants that are harder to kill than the ones containing a single fault. Within such context, mutants can fall into two categories: *(i)* mutants created by the injection of a single fault are called *first order mutants*, whereas *(ii)* mutants containing at least two faults are referred to as *higher order mutants*.

The concept of subsuming higher order mutants posits that since a subsuming higher order mutant denotes a subtle fault, it is harder to kill than its first order mutants from which it was created from (Jia and Harman, 2009). Consequently, first order mutants can be replaced by higher order mutants to reduce the number of mutants that need to be taken into consideration.

The savings that can be achieved by higher order mutation have been to a certain extent borne out by the results of Usaola et al. (2009). In their experiment, they emphasized a specific order of higher order mutant: the second order mutants (i.e., mutants containing two faults). Usaola et al. devised algorithms to consolidate first order mutants into second order ones. Their results indicate that second order mutants reduce the cost of mutant testing by approximately 50% while retaining much of the test effectiveness.

Langdon et al. (2009, 2010) employed multi-objective genetic programming to generate higher order mutants. According to them, the resulting higher order mutants are harder to kill than any first order mutant.

3.3.1.4 Mutant Clustering

Mutant clustering was first proposed by Hussain (2008). The idea involves selecting a subset of mutants using clustering algorithms (Xu and Wunsch, 2005). When using mutant clustering, after generating all first order mutants, a clustering algorithm is applied to classify the mutants into different clusters. The classification is as the following: the clustering algorithm assures that every mutant in a certain cluster is killed by a similar set of test cases. Therefore, aimed at reducing the computational cost associated with running mutants, only a small number of mutants is chosen from each cluster to compose the set of mutants that needs to be executed. The mutants that were not chosen are discarded.

Hussain examined two clustering algorithms: *(i)* k-means and *(ii)* agglomerative clustering. In his experiment, these algorithms were compared with greedy and random reduction strategies. The results indicate that mutant clustering selects fewer mutants without negatively affecting the mutation score. Ji et al. (2009) took mutant clustering a step further by using a domain reduction technique that obviates the need to execute all mutants.

3.3.2 Reduction of the Execution Cost

Even with automated tool support, one of the hurdles that have been hindering mutation testing from becoming a practical testing technique is the elevated computational cost associated with running the large number of mutants against a test set. Thus, aimed at dealing with this high computational cost, a research thrust in the area involves optimizing the execution process. The next subsections give an overview of the three approaches that have been used to boost the execution of mutants and mention several studies that explore each of these approaches.

3.3.2.1 The Spectrum of Mutation Testing Techniques: Strong, Firm, and Weak Mutation Testing

Mutation testing techniques can be classified according to when they evaluate the outcome to determine whether a mutant was killed (Jia and Harman, 2011). In the original technique, proposed by DeMillo et al. (1978), a mutant is said to be killed when it produces an outcome that is different from the one produced by the original program. When applying this technique, the outcomes are compared in the end of the execution process. Aimed at optimizing the execution of the original technique (hereafter referred to as *strong mutation*), a less computationally expensive as well as less stringent type of mutation testing, called *weak mutation*, was introduced by Howden (1982).

In weak mutation, the emphasis is shifted from entire programs to elementary components. It is assumed that a program P is comprised of a set of components $C = \{\mathrm{c}_1, \ldots, \mathrm{c}_n\}$. Such components can be of the following five types: variable assignment, variable reference, arithmetic expression, relational expression, and boolean expression (Howden, 1982). In essence, the idea is to create mutants by modifying these components: given a component c_m, a mutant m is created by changing c_m. Similarly, m is said to be killed if on at least one execution it yields an outcome different from c_m. Consequently, rather than checking mutants after the execution of the entire program, as is the case with strong mutation, they are checked immediately after the execution point of the mutated component (Jia and Harman, 2011). Actually, in weak mutation, execution is stopped after the mutation and if infection has occurred, the mutant is marked dead.

The definition of weakly killing a mutant encompasses only reachability and infection (Section 3.2). That is, the infected state does not need to propagate to the output. A mutant is killed when it produces a different intermediate state. Thus, it is possible to distinguish more than one mutant component in a single program execution; which is in stark contrast to strong mutation where, usually, after executing each program to completion only one mutant may be killed.

Offutt and Lee (1991, 1994) improved Howden's definition of components. They also refined how one should go about executing these components. The four types of execution that they came up with are the following: evaluation after the first execution of an expression (i.e., Ex-Weak/1), evaluation after the first execution of a statement (St-Weak/1), evaluation after executing a basic block (BB-Weak/1), and after n iterations of a basic block within a loop (BB-Weak/N).

As Woodward (1990) remarks, the main drawback of weak mutation is that distinct components can yield outcomes different from the original program on different executions and yet events can combine to either result in an overall correct outcome to the mutated component or to the entire program execution. Since Howden's initial study, several empirical studies were carried out to evaluate the pros and cons of weak mutation (Jia and Harman, 2011).

Girgis and Woodward (1985) developed a weak mutation tool for Fortran 77 programs. They implemented an analytical weak mutation tool in which the mutants are killed by probing the internal state of the program under test. In their experiment, they took into account four of the Howden's five elementary program components. Their results indicate that weak mutation is less computationally expensive than strong mutation. Similar conclusions were drawn from the results of the experiments conducted by Marick (1991).

Horgan and Mathur (1990) presented a theoretical proof of weak mutation. According to them, a test suite generated through the application of weak mutation can also be expected to be as effective as one created via strong mutation.

Offutt and Lee (1991, 1994) implemented a weak mutation system named Leonardo (Looking at Expected Output Not After Return but During Operation) that uses the 22 Mothra mutation operators rather than Howden's five elementary program components. Similarly to the results of Girgis and Woodward (1985), Horgan and Mathur (1990), and Marick (1991), the experimental results of Offutt and Lee suggest that weak mutation is an effective alternative to strong mutation.

Woodward and Halewood (1988) introduced *firm mutation* as a mutation testing strategy that falls somewhere in the middle of the two extreme ends of a spectrum of mutation techniques, namely, strong and weak mutation testing. Firm mutation has to do with performing mutation testing on partial executions of fragments of the program under test. For instance, given a statement within a loop that might be executed more than once in a certain execution, applying firm mutation entails changing such a statement on at least one execution of the loop. Therefore, a mutation is introduced into the program and persists for one or more executions but not for the entire execution. Figure 3.2 gives an overview of firm mutation. In Figure 3.2, t_{change} and t_{undo} represent the duration of the change, which is at least as long as the execution of a single statement. Figure 3.2 can also be used to put weak and strong mutation into perspective (Woodward, 1993). In weak mutation, t_{change} and t_{undo} take place immediately before and after the execution of a component. Strong mutation is when t_{change} and t_{undo} occur before and after the execution of the entire program.

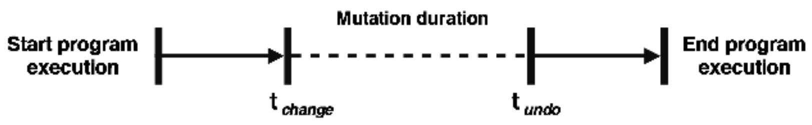


Figure 3.2: Firm mutation. The moment at which the fault is introduced into the program under test is denoted by t_{change}, and t_{undo} represents the moment when the change is reversed and the outcome is compared. This figure is adapted from Woodward (1993).

The advantages and disadvantages of firm mutation have yet be gauged (Woodward, 1993). According to Halewood and Woodward (1993), since firm mutation demands a greater degree of control over the mutation process and the execution environment itself, it will fit in well as a feature of an interpretive, interactive debugging environment. However, as of the time of this writing, there are no available implementation of such mutation testing strategy.

3.3.2.2 Run-time Optimization Techniques

Early implementations of mutation testing systems relied on interpretation (Offutt and King, 1987; King and Offutt, 1991). Due to the fact that the interpreter is the locus of control within such systems (Scott, 2009), Jia and Harman (2011) classified them as *interpreter-based optimization techniques.* Likewise, as stated by Jia and Harman, one of the main costs of interpreter-based techniques stems from interpretation. Based on the fact that the performance of interpretive systems depends heavily on the chosen HLR or DIR (Section 2.3.4.2) (Piumarta and Riccardi, 1998), several studies have tried to optimize the interpreter-based technique by translating the program source code into an intermediate form (Offutt and King, 1987; King and Offutt, 1991). However, even when translating the source code into an internal representation and performing mutation and interpretation at this intermediate level, interpretation itself does not scale well for large programs. Alternative run-time optimization techniques have therefore been sought.

As discussed in Chapter 2, compilation is a faster approach to implementing programming languages because it is devoid of the inefficiencies inherent to interpretation (Wilhelm and Seidl, 2011). Owing to this fact, researchers have been capitalizing on compilers to boost mutation testing. According to Jia and Harman (2011), the *compiler-based technique* is one of the most common ways of implementing mutation testing. A notable example of tool that builds on a compiler to implement an integrated mutation testing environment is Proteum (Delamaro and Maldonado, 1996). Proteum creates each mutant by modifying the source of the original program. Then, it proceeds to compile, link, and run mutants. Another similar example of compiler-based tool is Proteum/AJ (Ferrari et al., 2011), which supports the application of mutation testing to AspectJ programs. However, differently from Proteum, Proteum/AJ performs two compile-time steps for each mutant: compilation and weaving. As pointed out by Ammann and Offutt (2008), this approach is not well suited to small programs because the time spent compiling and linking or weaving them might greatly exceed the time they take to execute. Furthermore, very large programs can render this approach impractical because they result in compilation bottlenecks (Byoungju and Mathur, 1993).

As mentioned in Section 3.2, mutants differ from the original program only by a single minor syntactic modification. Hence, the compiler-based technique entails redundant compilation cost, as each mutant is compiled individually. To circumvent this drawback, Demillo et al. (1991) proposed the *compiler-integrated technique*. This technique involves modifying a compiler so that its output from the original program contains the following: *(i)* executable code for the original program and *(ii)* a set of patches for mutants. These patches have information on how to convert the initial executable code into an executable code that includes every mutant. By adding the mutants to the original program on the fly, this technique reduces the redundant cost stemmed from compiling each mutant separately.

The *mutant schema generation technique* (Untch, 1992; Untch et al., 1993) is also geared towards reducing the cost of compiling mutants individually. This technique is based on the program schema technique, which was proposed by Baruch and Katz (1988). Using the mutant schema generation, the underlying compiler is modified to encode all mutants into one program, called *metamutant*. The same compiler then is used to translate the metamutant (Ammann and Offutt, 2008). Given that all programs are included in a single file, a mechanism to drive the execution of the program and its mutants is also incorporated into the schema. Compiling all mutants in one compilation operation results in up-front savings, but the compiled metamutant runs slightly slower than a regular mutant. Several tools implement mutant schema for mutants generation: muJava (Ma et al., 2005), Javalanche (Schuler and Zeller, 2009), and Bacterio (Mateo et al., 2010).

Mateo and Usaola (2012) propose an improvement over mutation schema. Their improvement, called MUSIC (mutant schema improved with extra code), is able to determine when a given mutant must not be executed, thereby reducing the number of executions required. Results from an empirical evaluation show that the number of executions can be reduced by around 77%. However, the execution time spent by the original program increases around 56%. Moreover, according to Mateo and Usaola, their improvement is able to identify infinite loops more effectively then other techniques based on timeout.

Ma et al. (2005) came up with the bytecode translation technique. In this technique, mutants are not generated from the source code of the original program. Rather, mutants are created from the intermediate representation of the original program. Therefore, similarly to the aforementioned techniques, the compilation step is skipped for each mutant.

The execution environment simply fetches and executes the mutants in their intermediate representation. An advantage of this technique is that it can handle off-the-shelf programs whose source code is not available (Ammann and Offutt, 2008; Jia and Harman, 2011). This technique has been successfully implemented in Java (Offutt et al., 2004; Ma et al., 2005, 2006; Schuler et al., 2009). Nevertheless, not all programming languages support the manipulation of their intermediate representation.

Bogacki and Walter (2006, 2007) devised an alternative approach aimed to reduce compilation cost, called *aspect-oriented mutation*. Their approach consists in introducing aspect patches whose purpose is to capture the output of methods on the fly. Due to the introduction of these aspect patches, each method is executed twice. During the first execution, the aspect patch wrapping that method invocation obtains the result and context of the original method. Afterwards, in the second execution, mutants are generated and run. This obviates the need for compiling each mutant. Bogacki and Walter also compared their prototype tool with Jester (Moore, 2013), which is a strong mutation testing tool for Java.

3.4 Mutation Testing Tools

The reliance of mutation testing on automation was evident since the technique was proposed. Given that, many tools have been developed since the late 70's. Some of the most widely used tools have already been mentioned in this chapter, e.g., Proteum and muJava. According to Jia and Harman (2011), Mothra and Proteum were the first academic tools able to handle small, real-world programs not just toy programs. Consequently, both tools were widely studied. Many advances in mutation testing were initially experimented using these two tools. Jia and Harman also point out that after the first mutation workshop was held, there has been an increase in the number of academic mutation tools. Since then, much of the research described in this chapter has been translated into tools. A non-comprehensive list of academic tools is shown in Table 3.1. As shown in Table 3.1, the languages that have received the most attention so far are C and Java. Almost all tools listed in Table 3.1 automate strong mutation. Interestingly, the only tool designed for a dynamically typed language is SMutant. Further, the only four tools that implement concurrency-related mutation operators are ExMAn (EXperimental Mutation ANalysis), Javalanche, MuTMuT (MUtation Testing of MUltiThreaded code), and Paraμ (which are further described in Chapter 5).

Table 3.1: A non-comprehensive list of academic mutation testing tools. The implementations are listed in alphabetical order.

Name	Language	Mutation Type	Reference
AjMutator	AspectJ	Strong	(Delamare et al., 2009)
Bacterio	Java	System-level (Weak/Strong)	(Mateo et al., 2013)
CREAM	C#	Strong	(Derezinska and Szustek, 2008)
ExMAn	C and Java	Strong	(Bradbury et al., 2006a)
JavaMut	Java	Strong	(Chevalley and Thévenod-Fosse, 2003)
Javalanche	Java	Strong	(Schuler and Zeller, 2009)
Jester	Java	Strong	(Moore, 2013)
Judy	Java	Strong	(Madeyski and Radyk, 2010)
Jumble	Java	Strong	(Irvine et al., 2007)
MILU	C	Strong (Higher Order)	(Jia and Harman, 2008)
Mothra	Fortran	Weak/Strong	(DeMillo et al., 1988)
muJava	Java	Strong	(Ma et al., 2005)
MuTMuT	Java	Strong	(Gligoric et al., 2013a)
Paraμ	Java	Strong (Higher Order)	(Madiraju and Namin, 2011)
PIMS	Fortran	Strong	(Budd et al., 1978)
Proteum	C	Strong	(Delamaro and Maldonado, 1996)
Proteum/AJ	AspectJ	Strong	(Ferrari et al., 2011)
SMutant	Smalltalk	Strong	(Gligoric et al., 2011)
Testooj	Java	Strong	(Usaola et al., 2007)

3.5 Concluding Remarks

This chapter provided an introduction to mutation testing. Since the origin of the technique, research into mutation testing has come a long way, spawning several research thrusts. In particular, ways to optimize mutation testing have spurred a growing interest. A recent survey, carried out by Jia and Harman (2011), points out that research in reducing the cost of the mutation testing process has been drawing considerable attention. As this chapter forms the conceptual basis for the next chapter (which presents our approach to cost reduction and its implementation), special emphasis was given to approximation and cost reduction techniques. Moreover, some tools were highlighted.

As outlined in this chapter, there has been some overlap between the research communities working on designing programming languages and those interested in speeding up mutation testing. The Fortran interpreter developed by Offutt and King (1987) represents one of the first forays into taking advantage of execution environments to enable mutation testing. Since then, interpreters and compilers have been augmented in order to enhance the performance of mutant generation and execution. A quintessential example of language design idea that was adapted to boost mutation testing is the *program schema technique* (Baruch and Katz, 1988), which in the parlance of mutation testing is called mutant schema generation. However, to the best of our knowledge, the proof-of-concept implementation described in the following chapter is the first to take advantage of the benefits provided by a modern execution environment (e.g., JIT compilation and GC) to boost mutation testing.

Chapter 4

Speeding Up Weak Mutation Testing Through Harnessing HLL VMs

As mentioned in the previous chapter, one of the hurdles that have been preventing mutation testing from becoming more widespread is its elevated computational cost. To overcome the high cost of executing mutants, many approaches have been devised. As discussed in Section 3.3.2.2, several techniques have tried to boost mutation testing mainly by extending interpretive execution environments or modifying compilers. Similarly to the approaches mentioned in Section 3.3.2.2, the novel approach presented in this chapter can be seen as a run-time optimization technique specialized for mutation testing. The research herein described suggests that mutation analysis can be further sped up by being implemented within a modern HLL VM.

To evaluate this idea, we augmented a managed execution environment by adding support for mutation testing activities. We used the HLL VM that is by far the most widely used within academic settings (Durelli et al., 2010), the JVM. The implementation used is Maxine VM, which was discussed in-depth in Section 2.6. Instead of the more expensive strong variant of mutation, we use the cheaper approximation, weak mutation.

This chapter presents our proof-of-concept, VM-integrated mutation testing environment. The novelty of this approach is not the supporting technologies and tools, but rather their integration into a cutting-edge execution environment. Moreover, it also uses a novel way to implement weak mutation testing.

The remainder of this chapter is organized as follows. Section 4.1 outlines the HLL VM-based mutation analysis implementation and Section 4.2 summarizes the experimental results, which give some insight into possible performance improvements. Section 4.3 discusses the shortcomings of our implementation. In Section 4.4 some related work is discussed and Section 4.5 suggests future work and makes concluding remarks.

4.1 Proof-of-concept Implementation: Harnessing an HLL VM to Support Weak Mutation Testing

We extended Maxine VM (Section 2.6) to demonstrate the feasibility of developing a *mutation-aware JVM*. Our proof-of-concept implementation automates three key steps of mutation: execution of the original program, mutant execution, and mutant results comparison. Our mutation-aware HLL VM implements weak mutation testing (Subsection 3.3.2.1) and emphasizes individual methods. Java methods are the fundamental unit for encapsulating behavior and most JVMs are built to optimize their execution. Throughout this chapter we call methods in the program under test *originals*, and methods that have been modified for mutation *mutants*.

When the implementation reaches an original method invocation it prepares to execute all mutant methods related to the underlying original method. The program state is first saved, then the original method is executed. Then, each mutant method is called as a separate thread with a copy of the program state. The main program thread is held until all mutants are run. After finishing, the results of the mutant methods are compared with the results of the original method. As mentioned before, our implementation focuses on unit testing, so the results of method execution are defined to be the return value, which return location was used, instance variables of the class that the method referenced, and other local variables in the method. Details about saving the state are given in the following subsection.

When an original method is first invoked, it is pre-processed before being compiled and run by the JIT compiler. The pre-processing instruments the method to obtain a snapshot of the context before and after invocation. As broadly summarized in Figure 4.1, the instrumentation also transfers control to the entity responsible for triggering the execution, handling unexpected behavior, and analyzing mutant methods (i.e., *controller*). Original methods are instrumented at the beginning and at each of their return locations. Figure 4.1 outlines the pre-processing that takes place before an original method is JIT-compiled: the instrumented sections and code that is invoked from within the inserted instrumented code are shown in gray shade. After the pre-processing, the modified original method is JIT-compiled and executed.

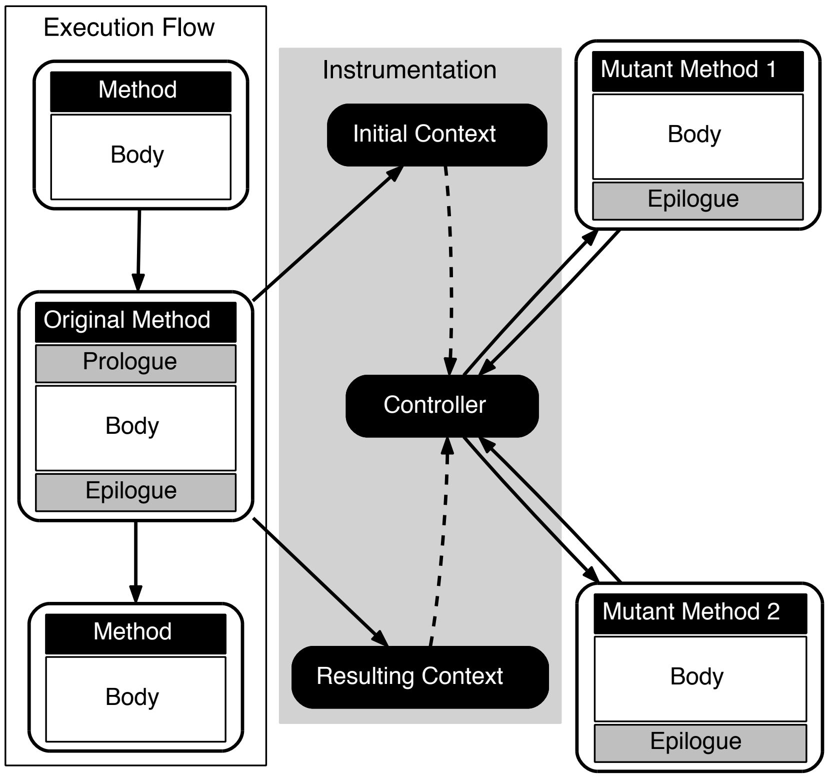


Figure 4.1: Overview of the pre-processing through which original methods undergo before being JIT-compiled.

As can be seen in Figure 4.1, in the presence of an already instrumented original method, the program's *calling sequence* (i.e., code executed immediately before and after a method call) is modified to invoke all mutant versions of the original method. Mutant methods are only instrumented at return locations, as shown in Figure 4.1, to capture the result of running the mutant.

Mutant methods are run in separate threads and results are compared with results from the original method immediately. If the mutant method has results that are different from the original method, the mutant is marked dead. The program under test resumes execution (i.e., the thread executing the original method) after all the mutant methods finish. Comparing results at the end of the methods makes this a weak mutation approach.

A benefit of forking new threads is that the program up until a method call does not need to be repeated for every mutant. This implements the split-stream execution proposed by King and Offutt (1991). This is a significant execution savings. As with all other mutation systems, dead mutants are not re-executed. Also, the mutant method look-up is performed only once for each method. The look-up takes place just after the first execution of the original method. After this, the controller keeps track of the mutant methods, invoking the ones that are still alive at the end of executions of their respective original methods.

4.1.1 Prologues

As described in the previous section, original methods undergo pre-processing before being invoked for the first time. The pre-processing instruments methods' *prologues*, which are located before their bodies. The instrumentation extracts a copy of the *initial context* as illustrated in Figure 4.1. What specifically is saved depends on the method. A class method's context is just the parameters passed to it, if any. An instance method always has at least one parameter since the index 0 is reserved for the `this` pseudo variable. Thus, the contexts of instance methods contain the receiver and all parameters.

Algorithm 4.1 gives a high-level overview of the instrumentation code for copying the initial context. Line 2 gathers the initial context into an array with a conservative approach: instead of computing the minimal depth required for the method's operand stack, we increase its size by 5. Additionally, an internal representation for the original method, a string, is generated and added to the runtime constant pool as shown in line 3. Next, if the method is an instance one, the underlying object is deep copied into the initial context array (lines 5–9).

Next, an array is initialized with information on the method's parameters (line 10). Since each Java primitive type has specific instructions that operate on them, parameter information is used to decide which load opcode needs to be instrumented to copy each parameter in the context array. Although not shown in Algorithm 4.1, primitive types are converted to object wrapper classes before being added to the initial context array. In addition, type information is used to find the index of the next variable to be included in the initial context array. This is needed because, as mentioned in Section 2.4, some primitive types take up two slots, and attempts to load `double` types using inappropriate opcodes would lead to type checking problems. Reference types are deep copied as shown in line 15 before being added to the array.

Finally, in line 29, after collecting the initial context into an array, the instrumentation code transfers both the method's internal representation and the context array to the controller. The controller uses copies of the information to invoke each mutant method. Thus, mutant methods are invoked with the same context that was passed to the original method.

It is worth mentioning that copying only a method's parameters at times does not encompass all the context needed to execute mutant methods. However, by not copying a large portion of the heap for every mutated method, our approach decreases the amount of storage space required to implement weak mutation.

Algorithm 4.1 Prologue instrumentation.

```
 1: procedure PROLOGUE($method$)
 2:     $method.opStack.size \leftarrow method.opStack.size + 5$
 3:     add $method.ID$ to $runtimeConstantPool$
 4:     $i \leftarrow 0$
 5:     if $method.isInstance?$ then
 6:         load deep copy of $method.receiver$
 7:         add $copy$ to $context[i]$
 8:         $i \leftarrow i + 1$
 9:     end if
10:     $parameters \leftarrow method.parameters$
11:     if $parameters.length \neq 0$ then
12:         for each $p$ in $parameters$ do
13:             if $parameters[i]$ is category 1 then
14:                 if $parameters[i]$ is reference then
15:                     load deep copy of $parameters[i]$
16:                     add $copy$ to $context[i]$
17:                 else
18:                     load $parameters[i]$
19:                     add $value$ to $context[i]$
20:                 end if
21:                 $i \leftarrow i + 1$
22:             else
23:                 load $parameters[i]$, $parameters[i + 1]$
24:                 add $value$ to $context[i]$, $context[i + 1]$
25:                 $i \leftarrow i + 2$
26:             end if
27:         end for
28:     end if
29:     transferToController ($method.ID, context$)
30: end procedure
```

4.1.2 Epilogues

Inserting epilogue instrumentation code is more complex than inserting prologues. It entails obtaining information on both the types of variables and their liveness. Each method has only one prologue, but can have multiple epilogues; one at each return site. For example, the method shown in Listing 4.1 has three return locations (i.e., ❶, ❷, and ❸), thus three epilogues are needed. Although `void` methods might not have explicit `return` statements, when they are rendered into bytecodes, opcodes of the family *xreturn* are inserted at their return locations. We use this to implement epilogues.

Our implementation discovers return locations by overriding all *xreturn* methods in Maxine VM bytecode pre-processor. These methods are invoked whenever a return opcode is found in the bytecode stream. Another technical hurdle was that while it is possible to know how large the local variable array is at a return location, knowing which variables are live and what their types are required further analysis.[1] We coped with this issue by modifying Maxine VM bytecode verifier to capture the state at return locations.

Listing 4.1: Example method with multiple return locations.

```
 1  public static double charge(int seasonCode, double amount) {
 2    if (seasonCode == SUMMER) {
 3      double summerRate = getSummerRate()-10.9;
 4      return amount + summerRate; ❶
 5    } else if (seasonCode == NEW_YEARS_DAY) {
 6      int multiplyBy = 2;
 7      return amount * multiplyBy; ❷
 8    } else {
 9      return amount;              ❸
10    }
11  }
```

To illustrate the dataflow analysis used to uncover which variables make up the resulting context, consider the example method shown in Listing 4.1. Before line 2 is executed, the set of local variables comprises just the method's parameters as shown in Figure 4.2(a). If the condition in line 2 evaluates to `true`, the statements in lines 3 and 4 are executed. During their execution, a variable of type `double` is created and stored in the local variables set, using two slots as depicted in Figure 4.2(b). Therefore, this variable is *live* at the return location ❶. If the condition in line 2 evaluates to `false` and the one in line 5 evaluates to `true`, an `int` is created and stored in the local variables set. As a result, the array of local variable would be as depicted in Figure 4.2(c) at return location ❷. Finally, if neither of the previous conditions is `true`, the else in line 9 (return location ❸) is executed and none of the previously mentioned variables are created. In such case, the set of local variables would be as depicted in Figure 4.2(a); the context has only two variables, `seasonCode` and `amount`. If `charge` were an instance method rather than static, the resulting context would also contain a reference to the instance upon which the method operates.

An additional ability is that when a mutant method makes an abnormal return by throwing an exception that is not handled in its body nor indicated in its signature, the

[1]Java files compiled with the `-g` option have debug data that could have been used to determine the liveness and type of variables. For flexibility, we decided not to impose such a constraint.

exception is handled by the controller. Since in this case the exception is thrown by a method inside a different thread, the thread running the controller has no access to the stack associated with the terminated thread. Therefore, our implementation performs no *"postmortem examination"* of the contents of the terminated thread and the resulting context is the exception in question.

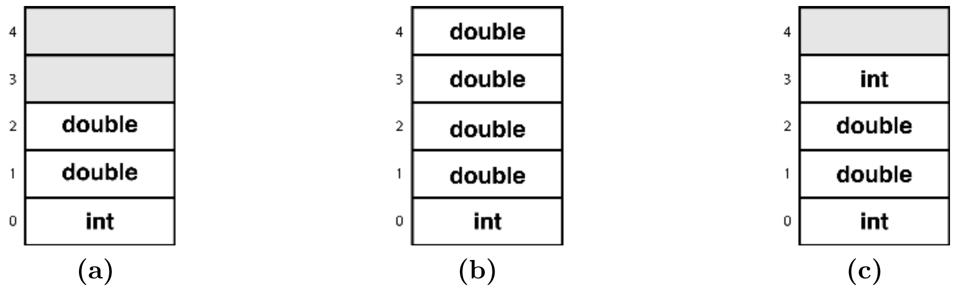


Figure 4.2: The three possible layouts of the array of local variables at each return location. Initially, only three slots are taken up by the method's parameters as depicted in (a). Afterwards, depending on which condition evaluates to `true`, the array of local variables may be either completely (b) or partially taken up as shown in (c) and (a).

4.2 Experiment Setup

This section describes the experiment setup used to gauge the performance of the VM-integrated mutation testing system. Specifically, we investigated the following research question:

> **RQ$_1$:** How much of the computational cost of mutation testing can be reduced by implementing it as an integral part of a managed execution environment and weakly killing mutants?

To evaluate this question, we compared our implementation with a conventional mutation testing tool for Java, muJava (Ma et al., 2005). A difference is that muJava implements strong mutation testing, but it is an easy to use Java mutation tool that has been widely used (Hu et al., 2011).

4.2.1 Goal Definition

We use the organization proposed by the Goal/Question/Metric (GQM) paradigm (Wohlin et al., 1999). GQM presents experimental goals in five parts: object of study, purpose, perspective, quality focus, and context.

- **Object of study:** The objects of study are our VM-integrated mutation testing implementation and muJava.
- **Purpose:** the purpose of this experiment is to characterize two approaches to automating mutation testing with respect to their computational costs. Specifically, we investigate whether VM-integrated mutation testing results in a marked improvement over the conventional approach (muJava). The experiment provides insight into how much speedup can be obtained by implementing mutation analysis within the managed runtime environment. It is also expected that the experimental results can be used to evaluate the impact of forking the execution of mutant methods in their own threads and weakly killing them have on the performance.
- **Perspective:** this experiment is run from the standpoint of a researcher.
- **Quality focus:** the primary effect under investigation is the computational cost as measured by execution time. The execution time of a program under test is defined as the time spent by the mutation testing system executing the program and all of its mutants against a test case.
- **Context:** this experiment was carried out using Maxine VM on a 2.1GHz Intel Core 2 Duo with 4GB of physical memory running Mac OS X 10.6.6. To reduce potential confounding variables, muJava was run within our modified version of Maxine VM. Since the current implementation is an academic prototype and the subject programs are not of industrial significance, this experiment is intended to give evidence of the efficiency and applicability of the technique solely in academic settings.

Our experiment can be summarized using the following template (Wohlin et al., 1999):

Analyze *our VM-integrated implementation and muJava*
for the purpose of *characterizing*
with respect to their *computational cost*
from the point of view of *the researcher*
in the context of *heterogeneous subject programs ranging from 11 to 103 lines of code.*

4.2.2 Hypothesis Formulation

Our research question ($\mathbf{RQ}_1$) was formalized into hypotheses so that statistical tests can be carried out:

Null hypothesis, H_0: there is no difference in efficiency between the two implementations (measured in terms of execution time) which can be formalized as:

$$\mathbf{H_0}: \mu_{\text{muJava}} = \mu_{\text{VM-integrated implementation}}$$

Alternative hypothesis, H_1: there is a significant difference in efficiency between the two implementations (measured in terms of execution time):

$$\mathbf{H_1}: \mu_{\text{muJava}} \neq \mu_{\text{VM-integrated implementation}}$$

4.2.3 Experiment Design

To verify our conjecture, we applied a standard design with one factor and two treatments (Wohlin et al., 1999). The main factor of the underlying experiment, an independent variable, is mutation testing. The treatments or levels of this factor are the two approaches to automating mutation testing: muJava and our VM-integrated implementation.

Before describing the particulars of this experiment, it is worth detailing the two treatments. We are comparing two different implementations of mutation testing. As mentioned, our mutation-aware HLL VM implements weak mutation testing and emphasizes individual methods. That is, instead of executing each mutant in a separated instance of the HLL VM and running each of them from the very beginning, during a single execution several mutant methods might be killed, depending on the result of their execution. This is possible because our implementation, upon reaching an original method that still has live mutant methods, saves the intermediate state of the execution. In addition, the VM-integrated implementation tries to further speed up the execution of such mutants by forking new threads to execute each of them. Thus, our mutation-aware HLL VM implements an improved split-stream execution (King and Offutt, 1991) wherein mutants are executed concurrently. By contrast, muJava implements strong mutation testing, executing mutants in the traditional way as presented in Section 3.2. Figure 4.3 graphically sums up these two different execution strategies.

In this experiment setup, the main dependent variable is execution time, which is defined as the time spent by a treatment to run a program and all of its mutants against a test case. We used six subject Java programs ranging from 11 to 103 lines of code. During the selection of these programs we focused on covering a broad class of applications, rather than placing too much emphasis on their complexity and size. Also, several of the subject programs have been studied elsewhere, making this study comparable with earlier studies. For example, `Triangle` implements the triangle classification algorithm, which has been broadly used as an example in the software testing literature (Ammann and Offutt, 2008).

Table 4.1 gives the number of executable Java statements and the number of mutants generated from each subject program.

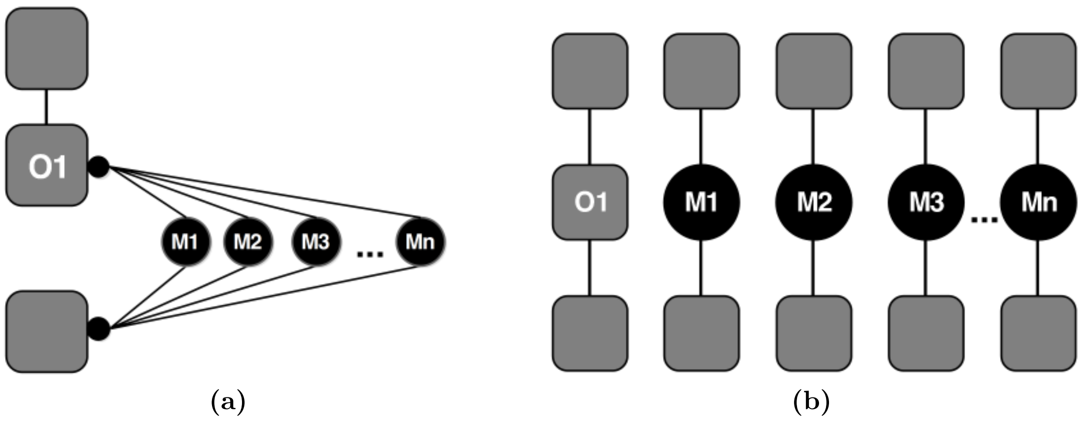


Figure 4.3: Execution strategies employed by the (a) VM-integrated implementation and (b) muJava. The gray rounded-edge squares represent parts of the original program (i.e., O1) and the black circles represent mutants (i.e., M1…Mn). As shown in (a), our VM-integrated implementation uses an improved split-stream execution strategy. By using this strategy, only one execution per test case is required. Such an execution evaluates all mutants in their own threads. In contrast, muJava's execution strategy, shown in (b), entails more than one execution to evaluate all mutants.

Table 4.1: Subject programs used in the experiment.

Subject Program Name	**Experimental Subject Programs**	
	Lines of Code*	**Number of Mutants**
`Fibonacci`	11	49
`ArrayMean`	13	38
`InsertionSort`	14	63
`ArrayIterator`	35	46
`KMP`	53	140
`Triangle`	103	316

*Physical lines of code (non-comment and non-blank lines).

All mutants were generated using muJava, which implements class and method-level (i.e., traditional) mutation operators. Because our interest is on testing methods, and the class-level mutation operators focus on the integration of classes, this experiment only used the method-level operators. The method-level operators are *selective* (Offutt et al., 1996a), that is, only the mutation operators that are most effective. The mutants were run on both treatments, which required minor modifications since muJava generates new `.class` (bytecode) and Java files for each mutant, whereas our implementation requires that all faulty versions of a method be placed in the same file as their original method. To adapt muJava mutants to the VM-integrated implementation, we collected all mutants of a method into the same class file, and gave them unique names corresponding to the mutant type (i.e., operator name) and a number. For example, ROR mutants for method `mean()` were named `mean$ROR_1()`, `mean$ROR_2()`, etc. These operations were mostly performed by a Ruby script.

For each program, we randomly generated 100 test cases. These tests were then executed against the mutants under both treatments. `ArrayMean` and `InsertionSort`, for instance, were executed using tests containing arrays of randomly-varying sizes (ranging from 0 to 1000) filled with random `double` values. Since we were interested solely in investigating the run-time of the two execution models, we did not try to generate mutation-adequate test sets. The execution time of each test case was calculated based on the mean of three executions. To deal with mutants that go into infinite loops, we set both treatments with a three-second timeout interval.

4.2.4 Analysis of Data

This section sets out the experimental findings. The analysis is divided into two subsections: (*i*) descriptive statistics and (*ii*) hypothesis testing.

4.2.4.1 Descriptive Statistics

This subsection provides descriptive statistics of the experimental data. Figure 4.4 charts the mean execution times for the subjects against the test sets. From Figure 4.4, it can be seen that the Maxine VM implementation outperforms muJava on all subjects, and significantly so for some. The speedup is less for some subjects, and it seems that the larger differences are when the subject program uses more computation (e.g., `InsertionSort` and `KMP`) and when the tests make several method calls (for example, `Triangle`, whose test cases comprise method calls to classify the underlying triangle and compute its perimeter and area).

Since the execution times diverge so much, particularly with muJava, we consider the median to be more a useful measure of central tendency than the mean. The median execution times are in Table 4.2, which shows a maximum difference of 95% for `Triangle`.

The data dispersion is shown in Table 4.3, which shows the standard deviations. These data show that muJava had larger standard deviation values than Maxine VM, which suggests that using muJava results in a high variability in execution times. Presumably, the VM-integrated implementation is more consistent because it requires only one execution per test case to evaluate all mutants (the forking, or split-stream approach). muJava, on the other hand, separately executes the original program and each live mutant for each test.

That is, in addition to the overhead of repeatedly loading new versions of the program, muJava also has to execute chunks of code that have already been executed until it reaches mutation points. Another advantage of the VM-integrated implementation is that its performance is not adversely affected when most of the mutants are killed due to timeout: each mutant executes in its own thread, thus mutants that end up in an endless loop do not affect the execution of other mutants. They are eventually preempted and killed by the controller. This feature resulted in significant savings for `InsertionSort`.

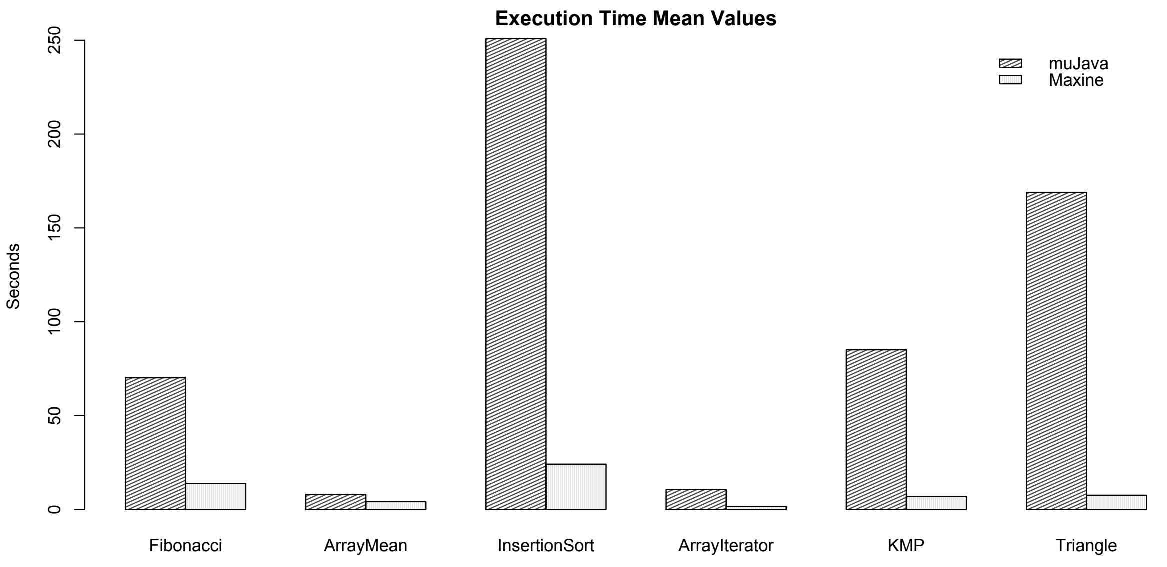


Figure 4.4: Execution time mean values for each subject program under each of the two treatments.

Table 4.2: Median of the execution times for each subject program under both treatments.

Median			
Subject Program Name	**Treatment**		
	muJava	**Maxine**	**% Diff**
`Fibonacci`	63.49s	13.77s	78.31%
`ArrayMean`	7.48s	4.16s	44.39%
`InsertionSort`	225.90s	23.97s	89.39%
`ArrayIterator`	10.94s	1.56s	85.74%
`KMP`	58.85s	6.77s	88.50%
`Triangle`	162.90s	7.35s	95.49%

Table 4.3: Standard deviations of the execution time for each subject program under each of the two treatments.

Standard Deviation		
Subject Program Name	**Treatment**	
	muJava	**Maxine**
`Fibonacci`	34.26s	0.50s
`ArrayMean`	3.36s	0.24s
`InsertionSort`	124.67s	0.48s
`ArrayIterator`	1.66s	0.31s
`KMP`	62.82s	0.42s
`Triangle`	105.62s	1.29s

Figure 4.5 summarizes the sampled run-time data, providing an overview of the significant savings in execution that can be achieved by our VM-integrated implementation. Besides achieving significant savings, it can be seen from observing Figure 4.5 that our implementation also performs more consistently.

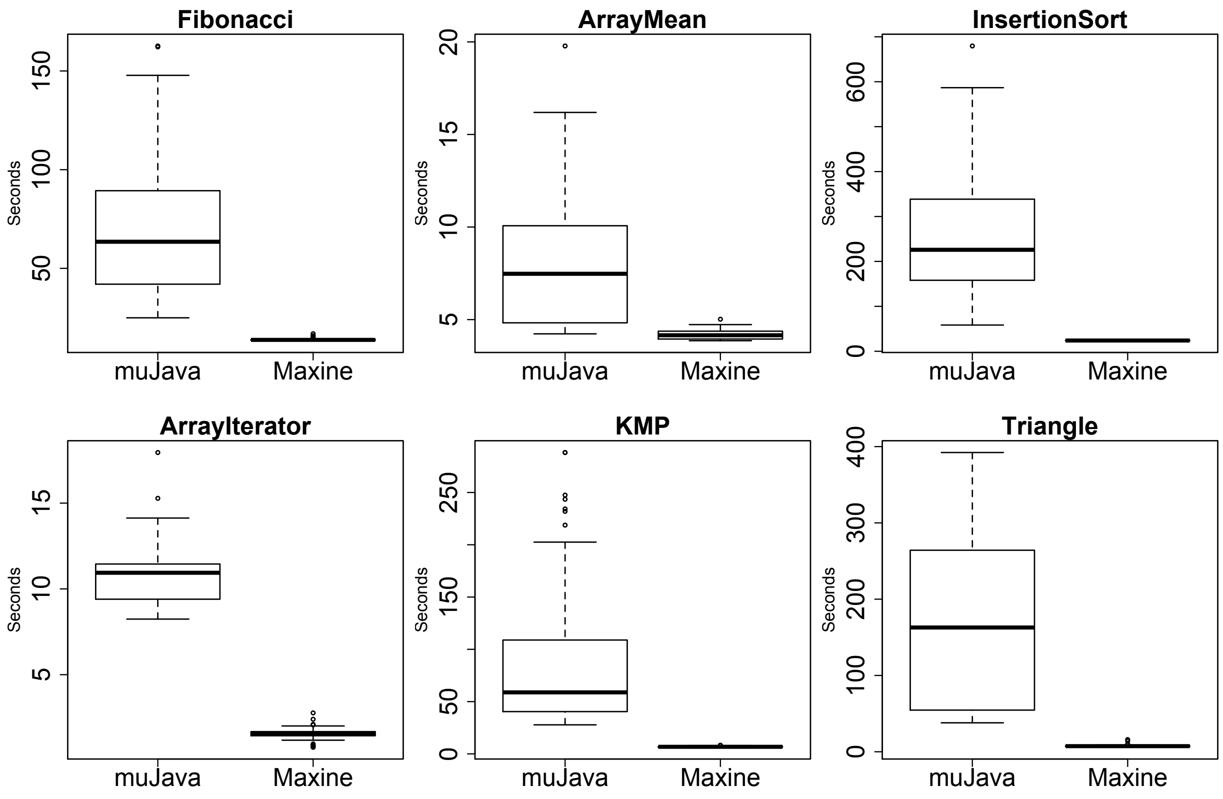


Figure 4.5: Boxplots of the execution times for each subject program under each treatment. muJava shows significant execution time variability.

4.2.4.2 Hypothesis Testing

Since some statistical tests only apply if the population follows a normal distribution, before choosing a statistical test we examined whether our sample data departs from linearity. We used Quantile to Quantile (Q-Q) plots (Dodge, 2009) as shown in Figure 4.6, which show that most sets of data depart from linearity, indicating the non-normality of the samples. These plots also show the presence of outliers. Instead of depending solely upon these plots, all distributions were assessed for normality using the Shapiro–Wilk test (Johnson and Bhattacharyya, 2009; Teetor, 2011). According to this test, $p < 0.05$ suggests that the population is likely not normally distributed, whereas $p > 0.05$ indicates that there is no such evidence. The results in Table 4.4 suggest that it is unlikely that these samples came from a normal population. Thus, we used a non-parametric test, the Wilcoxon signed-rank test (Johnson and Bhattacharyya, 2009).

Table 4.4: The significance of the p-value of the Shapiro–Wilk test for departures from normality.

Shapiro–Wilk test of normality (p-value)		
Subject Program Name	**Treatment**	
	muJava	**Maxine**
`Fibonacci`	*** (3.563e-05)	*** (2.185e-14)
`ArrayMean`	*** (4.749e-06)	*** (6.817e-07)
`InsertionSort`	*** (0.00041)	*** (1.247e-06)
`ArrayIterator`	*** (1.464e-05)	*** (9.831e-05)
`KMP`	*** (1.877e-10)	*** (1.608e-07)
`Triangle`	*** (2.114e-07)	*** (8.952e-15)
* p < 0.05; ** p < 0.01; *** p < 0.001.		

Table 4.5 shows the results of hypothesis testing with a significance level of 1%. Based on these data, we conclude there is considerable difference between the means of the two treatments. We were able to reject $\mathbf{H}_0$ at 1% significance level in all cases. All the p-values are very close to zero, as shown in Table 4.5, which further emphasizes that the VM-integrated implementation performs significantly better than muJava.

From observing the confidence intervals shown in Table 4.5 it can be seen that the VM-integrated implementation led to substantial savings in execution time in most cases. For instance, savings of approximately 187.53 seconds for `InsertionSort` and 134.14 seconds for `Triangle` were achieved. The worst-case was `ArrayMean`, for which speedups of only about 2.74 seconds were achieved.

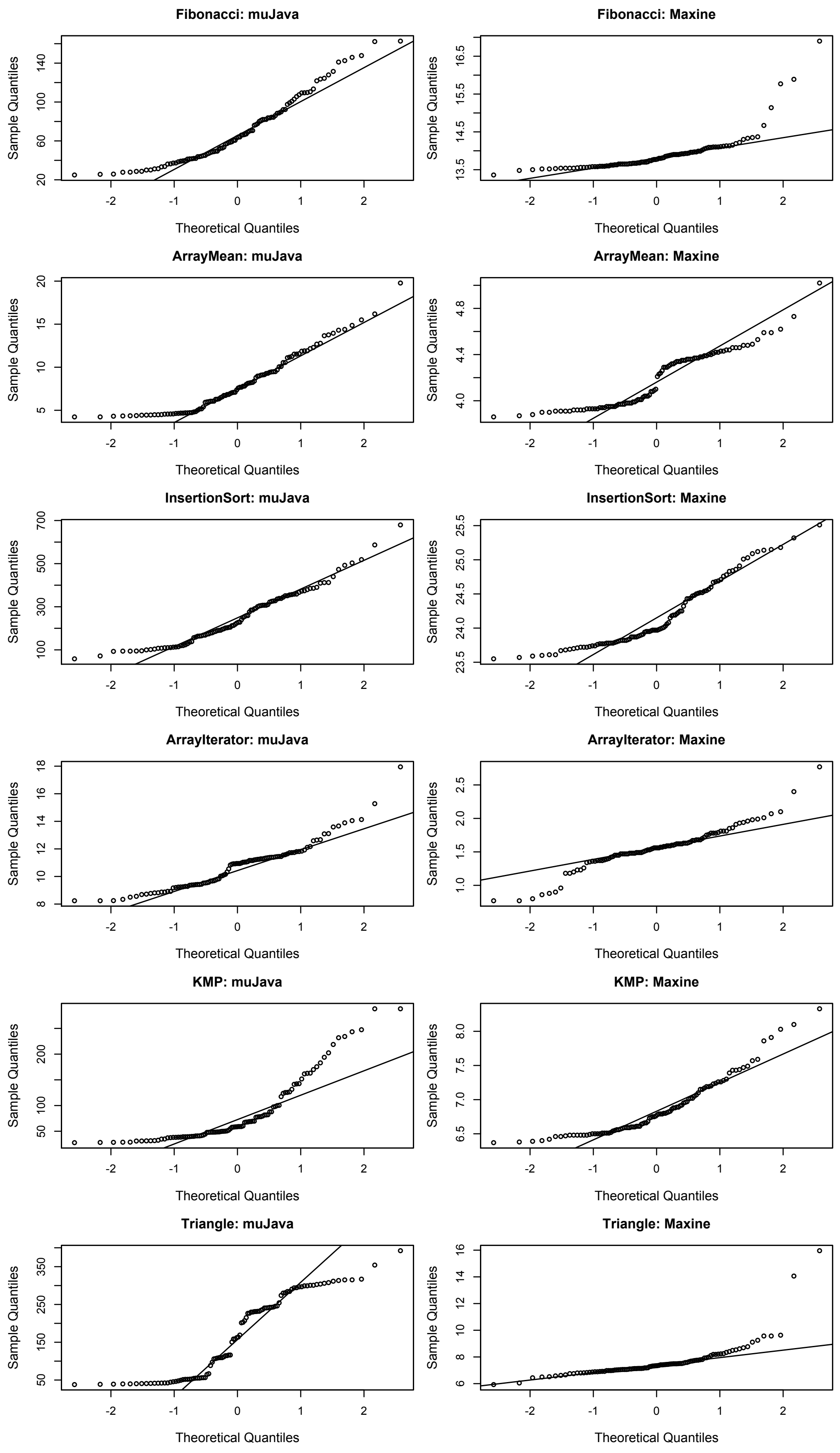


Figure 4.6: Normal probability plots of execution times for each subject program under each treatment.

Table 4.5: Testing of hypotheses for each subject program.

	Hypothesis Testing					
Subject Program Name	**T^+**	**T^-**	**V**	**P-value**	**Confidence Interval**	**Comment**
`Fibonacci`	5050	0	5050	< 2.2e-16*	44.13 to 63.53	We can reject $\mathbf{H}_0$ at the 1% significance level.
`ArrayMean`	5036	14	5036	< 2.2e-16*	2.74 to 4.62	We can reject $\mathbf{H}_0$ at the 1% significance level.
`InsertionSort`	5050	0	5050	< 2.2e-16*	187.53 to 252.69	We can reject $\mathbf{H}_0$ at the 1% significance level.
`ArrayIterator`	5050	0	5050	< 2.2e-16*	8.67 to 9.50	We can reject $\mathbf{H}_0$ at the 1% significance level.
`KMP`	5050	0	5050	< 2.2e-16*	50.70 to 87.51	We can reject $\mathbf{H}_0$ at the 1% significance level.
`Triangle`	5050	0	5050	< 2.2e-16*	134.14 to 186.95	We can reject $\mathbf{H}_0$ at the 1% significance level.

*2.2e-16 means $2.2X10{-}16 = 0.00000000000000022$. This is the smallest non-zero number R can handle (Teetor, 2011). Hence, the p-values shown above are not very accurate, however, they are definitely small. Therefore, we can confidently reject $\mathbf{H}_0$ in all cases.

4.2.5 Threats to Validity

The lack of representativeness of the subject programs may pose a threat to external validity. Unfortunately, this is a problem with virtually all software engineering research, since we have no theory to tell us how to form a representative sample of software. Apart from not being of industrial significance, another potential threat to the external validity is that the investigated programs do not differ considerably in size and complexity. To partially ameliorate that potential threat, the subjects were chosen to cover a broad class of applications. Also, this experiment is intended to give some evidence of the efficiency and applicability of our implementation solely in academic settings.

The fact that the VM-integrated approach achieved more speedup with programs that had more computation and more method calls indicates any bias could be against the VM-integrated approach. That is, it may perform even better in industrial practice. Furthermore, since our study focuses on evaluating strategies mostly used for unit testing, the size of the subject programs should not invalidate the results.

A threat to construct validity stems from possible faults in both implementations. With regard to our VM-integrated implementation, we mitigated this threat by running a carefully designed test set against several small example programs. A factor that hindered the use of such test set to perform corrective regression testing was the slow turnaround time: changes to Maxine VM files required at least a five-minute wait for recompilation. muJava, has been extensively used within academic circles, so we surmise that this threat can be ruled out.

4.3 Known Limitations

Java programs can interoperate with non-Java programs through the Java Native Interface (JNI) (Husaini, 1997; Liang, 1999). From Maxine VM's perspective (as well as other JVM implementations), a call to a native method is different from a regular method call. The most striking difference is that the Java stack (described in Section 2.4.4) is not used. Instead, native methods use their own stack, called C stack (Liang, 1999; Lindholm and Yellin, 1999). As our implementation is build around Java stacks, it does not support Java methods containing calls to native methods.

4.4 Related Work

The goal of this research is similar to other efforts that attempt to reduce the computational cost of mutation analysis. Our approach differs from others because it embeds mutation analysis into the HLL VM execution engine and makes mutants first-class entities. Other studies have extended HLL VMs to reduce the overhead of other testing activities: *(i)* collecting coverage data and *(ii)* storing compiled code during test suite execution.

To address the former issue, Chilakamarri and Elbaum (2004) proposed to remove coverage probes after they have been executed. They implemented two strategies, collectively termed disposable coverage instrumentation, in the Kaffe JVM. The first strategy removes coverage probes after execution. The second strategy improve the first one by incorporating coverage data gathered from multiple instances of the modified JVM running the application under test, thereby avoiding the execution of probes that have already been covered by any of the concurrently running instances.

The second issue (storing seldom used native code during test suite execution) was approached by Kapfhammer et al. (2005). They modified the Jikes RVM to adaptively ascertain and unload rarely used native code from the heap.

4.5 Concluding Remarks

As discussed in Chapter 3, performance is still a major issue in mutation testing, which makes the problem addressed by our proof-of-concept implementation highly relevant. Our implementation capitalizes on features already present in a managed execution environment and its underlying virtual ISA to boost the performance of weak mutation analysis. Although other researchers have retrofitted software testing features into interpretive execution environments and a plethora of instrumenting systems have been implemented, to the best of our knowledge this is the first effort to incorporate the ideas of mutation testing into a full-fledged, JIT-enabled HLL VM.

In this chapter, we described the basic design and components of our proof-of-concept implementation, which builds on a contemporary JVM implementation. Our current implementation has been deliberately limited on first getting the concepts right, before any additional performance optimizations are to be considered. Nevertheless, according to our experiment results, it can be concluded that the VM-integrated approach to weak mutation outperforms a conventional strong mutation testing tool by a large margin in terms of execution time.

Given that each mutant method execution is fairly independent, the problem of speeding up the execution of such mutants lends itself very well to concurrency. As far as we know, our implementation is the first to exploit multithreading to boost weak mutation performance. This novel characteristic of our implementation yields marked execution savings by concurrently executing mutant methods. It proved to be particularly useful for speeding up the execution of mutants whose executions end up in infinite loops. Since our implementation executes each mutant method in its own thread, the mutants that become stuck in a loop do not affect the execution of others. This benefit offsets the overhead of the parallel infrastructure, which usually has to keep track of the forked threads.

Spawning a new thread for each mutant might not be an optimal approach. Therefore, our implementation allows for using a thread pool whose size can be specified via command line. Using approaches to sizing the thread pool properly, as the one described by Goetz et al. (2006), may yield better performance. For instance, the nature of the mutants (e.g., compute-intensive mutants) and the computing environment (e.g., number of available cores) are data that can be used to tune the thread pool according to its workload. However, more experiments need to be carried out to evince whether our fork-and-join model is scalable, that is, whether execution speed increases when more processors are available.

Future work stemming from this first experiment needs to examine how much of the achieved speed-up is due to multithreading and how much can be attributed to weak mutation. Furthermore, the subject programs we used to evaluate our implementation are not very representative: the largest subject program contains 103 lines of code. To investigate further and show evidence of the scalability of our implementation, we intend to carry out a follow-up experiment using more representative programs.

As mentioned in Chapter 2, Maxine VM features no interpreter. Rather, Maxine VM compiles all methods prior to execution, which implies that considerable amounts of machine code are created in the course of executing a program and its mutants. Machine code related to dead mutants become outdated: these methods will no longer be invoked through the course of the program's execution. Aimed at addressing the memory requirement issue, Maxine VM supports code eviction (Oracle Corporation, 2012). However, eviction is a *"stop-the-world"* operation, e.g., all threads are suspended during an eviction cycle. Hence, a program containing many mutants might cause several stop-the-world operations, resulting in potential performance loss.

Chapter

5

Testing of Concurrent Programs

Early on, most computers had a single core. Multicore computers, which have at least two cores, used to be quite expensive and only large data centers and scientific computing facilities could afford this sort of computer (Goetz et al., 2006). Due to advances in technology and decreasing hardware price, recently, multicore processors[1] have gone mainstream. This widespread adoption of multicore architectures has brought about significant changes from the standpoint of program development (Sutter and Larus, 2005). To leverage the benefits provided by these multicore architectures, software practitioners have been turning to the use of concurrency abstractions available in most OSes and contemporary programming languages (Adl-Tabatabai et al., 2006; Kim and Bond, 2009; Hand, 2012). Furthermore, concurrency has been attracting a great deal of attention from researchers and practitioners. Among other things, researchers have been trying to harness the computing power of multicore architectures by designing technologies that hide the complexity of these architectures, thereby allowing programmers to focus on algorithms rather than the intricacies of such architectures (Kim and Bond, 2009).

Despite its benefits, concurrency poses several challenges to testing and debugging. This chapter describes these challenges and how researchers have been trying to overcome them. By elucidating these topics, this chapter motivates our HLL VM-based approach to supporting the test of concurrent programs, which is described in Chapter 6. It is worth noting that the emphasis in this chapter is on providing background that is pertinent

[1]Multicore processors are also know as chip multiprocessors (Sutter and Larus, 2005; Stallings, 2011).

to understand our HLL VM-based approach, instead of giving a complete account of the issues of concurrency.

The remainder of this chapter is organized as follows. Section 5.1 provides background on multithreaded programs. The abstractions used to represent concurrency in the context of HLL are described next. This is followed by a discussion of the challenges that concurrency poses to testing and debugging. In Section 5.2 we outline some problems that may arise in concurrent programs. Section 5.3 focuses on techniques that have been used to detect those concurrency-related problems. The discussion is divided into two parts: the first covers static techniques and the second describes dynamic techniques. A discussion on how researchers have been investigating the application of mutation testing to concurrent programs is presented in Section 5.4. Concluding remarks are given in Section 5.5.

5.1 Background

The earliest computers had no OS and executed a single program at a time (Goetz et al., 2006; Stallings, 2011). Over time, in order to improve computational resource utilization, computers started to include OSes that allowed for the execution of more than one program at once. This was an important development towards allowing the execution of multiple programs in a concurrent fashion (Tanenbaum, 2007; Cantrill and Bonwick, 2008a,b). This rudimentary category of concurrency is called *logical concurrency* (Sebesta, 2012) because programs act as if there were multiple processors, when in reality programs are being run in an interleaved fashion on a single processor. To make this possible, OSes switch the available processor and its computational resources from program to program. When there is more than one processor and they are able to run several programs simultaneously the concurrency that takes place is called *physical concurrency* (Sebesta, 2012). In both cases, executing programs are commonly referred to as *processes* (Tanenbaum, 2007; Stallings, 2011).

Given that programs can be designed for platforms that support only logical concurrency as if there were multiple processors, the discussion in the remainder of this chapter applies to both sorts of concurrency. Thus, for brevity's sake, only the term concurrency will be used.

Often, processes have a single thread of control. The *single-threaded* approach implemented by some OSes is illustrated in Figure 5.1(a). Current OSes allow processes to have more than one thread of control, sharing the same address space and other resources (Tanenbaum, 2007). A process (or program) that has more than one thread of control is said to be *multithreaded* (Stallings, 2011; Sebesta, 2012). Figure 5.1(b) depicts a multithreaded approach. Naturally, OSes that implement such an approach can spawn any number of processes which, in turn, can have any number of threads assigned to them as depicted in Figure 5.1(c). Although most resources are available to all threads sharing the same address space, some resources are exclusively allocated to each thread. For example, each thread has its own stack as shown in Figure 5.1(d).

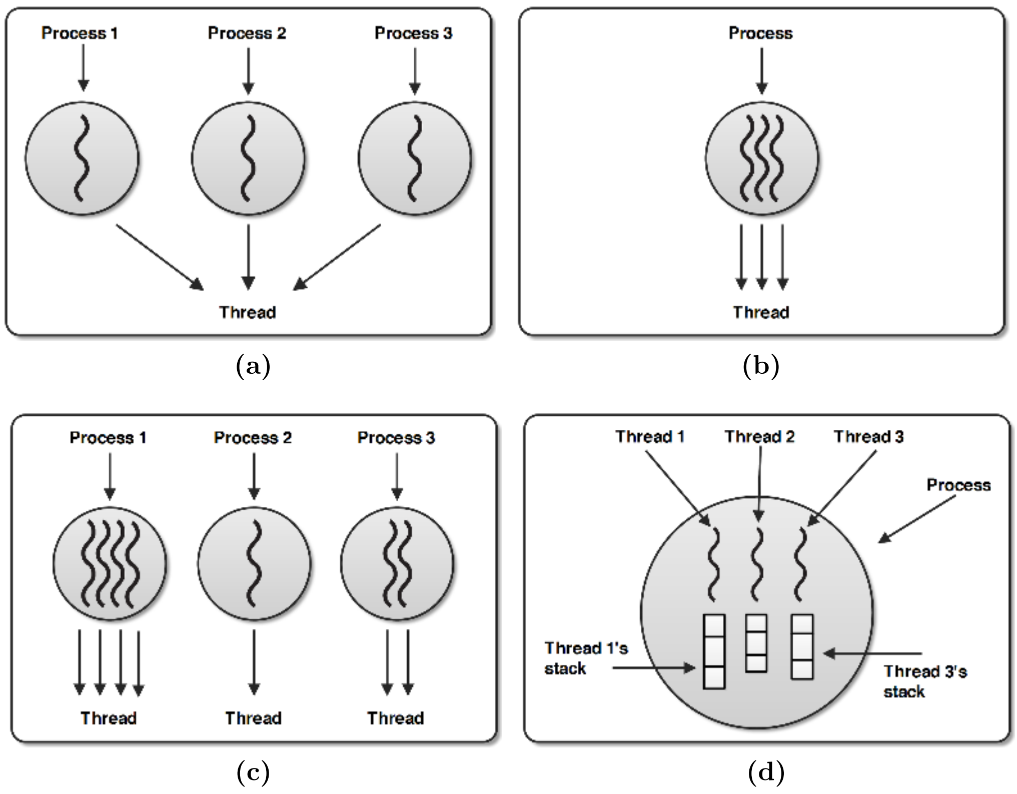


Figure 5.1: Single-threaded (a) and multithreaded (b) approaches. Each process can have multiple threads (c). Although global and some process-level resources are shared by threads, threads also have local resources, e.g., stack (d). These figures were adapted from Tanenbaum (2007).

The main benefit of having more than one thread of control is that multiple executions can take place within the same process environment. In a way, as pointed out by Tanenbaum (2007), this is similar to having multiple processes concurrently executing in a computer. The distinction is that threads share the resources within the address space of their respective process, while processes share the physical resources managed by the underlying OS (Ben-Ari, 2006). Because of the similarity between threads and processes, threads are sometimes referred to as *lightweight processes* (Tanenbaum, 2007; Stallings, 2011).

5.1.1 Concurrency in the Realm of Programming Languages

Ever since multithreaded OSes became available, language developers have been interested in taking advantage of concurrency. The theoretical groundwork for exploring concurrency at language level has been laid since the 60s (Ryder et al., 2005; Scott, 2009). Even early programming languages already included language-level support for concurrent programming. As noted by Ryder et al. (2005), the characteristics of certain programs spurred the introduction of concurrency in programming languages. Simula 67 (SIMple Universal LAnguage 67) (Dahl et al., 1968; Fraser et al., 2007), for instance, introduced concurrency to cope with the implementation of simulations. Coroutines were the construct used to represent concurrency in Simula 67 (Scott, 2009), they allowed quasi-parallel execution. Around the same time, Algol 68 (ALGOrithmic Language 1968) (Lindsey, 1993) was able to achieve true concurrency (i.e., physical concurrency) through the `par begin-end` construct. Ada (Sammet, 1986), which was designed in the mid 70s, was arguably the first HLL designed with constructs for concurrency that achieved widespread adoption (Ryder et al., 2005).

Along with abstractions to represent concurrency (e.g., coroutines and threads), language implementors also had to address two important design issues: *communication* and *synchronization* (Scott, 2009). Mechanisms of communication are needed because threads use information produced by other threads. Usually, imperative programs use *shared memory* for communication. Given that variables are easily accessible to threads in a shared-memory model, threads communicate with each other by writing to and reading from these shared variables. Synchronization is used to control the order in which threads run and access shared data. There are two sorts of synchronization: *cooperation synchronization* and *competition synchronization* (Sebesta, 2012).

Cooperation synchronization takes place when two threads, T_1 and T_2, are involved in an operation, and T_1 needs to wait for T_2 to complete part of the operation before it can start or resume its execution. Competition synchronization happens when two threads share data that cannot be used at the same time. For instance, T_1 needs to access shared resource ρ while T_2 is using it, T_1 must wait for T_2 to release the shared resource before accessing it. In other words, competition synchronization prevents two or more threads from accessing a shared resource at the same time, which could jeopardize the integrity of such resource (Sebesta, 2012). Therefore, synchronization can be viewed as a set of constraints on the ordering of events and access to shared data (Andrews and Schneider, 1983).

In an attempt to provide mutually exclusive access to shared data, Edsger Dijkstra came up with the concept of *semaphores* in the late 60s (Dijkstra, 1965). Dijkstra (2002) also showed how semaphores can be used to solve a number of competition synchronization problems. According to Sebesta (2012), semaphores can also be used to provide cooperation synchronization.

Generally, a semaphore is implemented using a counter and two associated operations, i.e., `P` and `V` (Reek, 2002). A semaphore plays the role of a lock: if the counter's value is non-negative, a thread is allowed to proceed by invoking `P` on the semaphore, which decrements the counter's value. Semantically, invoking `P` signals that the thread has been granted permission to proceed with its execution. `V` is invoked when a thread wants to signal that it is finished. Calling `V` increases the counter's value, allowing the other threads waiting on the semaphore to become eligible to resume. Semaphores whose counters are restricted to assume only two values (0 and 1 or `true` and `false`) are called *binary semaphores* (Sebesta, 2012), regular semaphores are sometimes referred to as *counting semaphores* (Scott, 2009). Semaphore-based implementations of synchronization appear in Algol-68 and Modula-3 (Scott, 2009).

Although widely used, semaphores are a low-level synchronization mechanism with drawbacks that make them prone to programming mistakes. For example, the programmer has to keep track of the calls to `P` and `V`, which appear scattered throughout programs. Consequently, it is hard to track them down for maintenance purposes. Moreover, leaving out a call to `V` can result in synchronization problems.

Dijkstra (1971) proposed *monitors* as a solution to these problems. The idea was further developed by Hansen (1973) and formalized by Hoare (1974). Monitors improve upon the concept of semaphores (Sebesta, 2012) by encapsulating shared data structures and their respective operations into a single unit.

A monitor is implemented as a module or object. A distintive feature of a monitor is that its methods are run in a mutually exclusive fashion: at any point in time at most one thread is executing any of its methods. This characteristic relieves the programmer from the burden of using `P` and `V` correctly (Scott, 2009). As pointed out by Ryder et al. (2005), a monitor can be seen as an abstract data type that has undergone modifications to fit in with the needs of a concurrent setting. The first HLL to include a monitor-based solution to synchronization was Concurrent Pascal (Hansen, 1996). By the mid 90s, Java took the monitor concept a step further by wrapping shared data into synchronized methods and blocks (the Java implementation of monitors is discussed in-depth in Chapter 6).

Despite the fact that the advances in concurrent programming stretch back to the 1960s, the explosion of interest in concurrency by both academia and industry members is a relatively recent occurrence. Apart from the availability of multicore computers, this growing interest can be ascribed to the proliferation of graphical and web-based applications, whose implementations often make extensive use of concurrent abstractions (Goetz et al., 2006; Scott, 2009). Currently, because of this surge in interest, most contemporary languages provide some sort of support for concurrency. Therefore, abstractions for representing concurrency in HLL abound. Nevertheless, threads have been the most widely used abstraction (Lee, 2006).

Even though great strides have been made on research and language-level support for concurrency, designing concurrent programs using the current iteration of concurrent technologies is still error-prone. To make matters worse, when it comes to concurrent programs, conventional testing and debugging are not suited to uncover and replicate problems. Next subsection summarizes the characteristics that make concurrent programs challenging for testing and debugging.

5.1.2 The Challenges of Testing and Debugging Concurrent Programs

Although concurrent programs afford opportunities for performance optimizations, they are also subject to problems that do not appear in their sequential counterparts. Nondeterministic behavior is one of the problems that make these programs notoriously complex to test. As Carver and Tai (1991) remark, the conventional testing aims to find faults by running programs against a test suite and then comparing the outcomes with the expected results. If any outcome deviates from the intended one, usually the same test data is used to reproduce the erroneous execution and gather debugging information. Such a debugging information is used to diagnose and ultimately fix the fault. After fixing the program under test, it is executed once again with the same test suite, making sure that the previously detected problem was corrected and that no new faults were introduced.

Due to the characteristics of concurrent programs, the straightforward approach of executing programs with a set of inputs in hopes of exposing faults is not adequate. During execution, concurrent programs undergo a series of *synchronization events* called *synchronization sequence* (Carver and Tai, 1991). Given the unpredictability of using nondeterministic synchronization constructs, multiple executions of a concurrent program are bound to exercise different synchronization sequences, which might lead to different outputs. To apply the conventional testing approach to concurrent programs, several issues that arise from the nondeterminism in concurrent programs need to be dealt with. For example, this nondeterminism leads to a nontrivial issue for test automation: it is complex to force a deterministic re-execution of a certain program statement or branch because testing tools do no try to prune away nondeterminism by controlling synchronization sequences (Long et al., 2003; Sen, 2007). Namely, by exerting no control over the program under test, testing tools might execute the same synchronization sequence many times (Edelstein et al., 2003). Because of this lack of control, conventional methods of debugging, such as using a debugging tool to set breakpoints and rerun the program under test, do not work properly for concurrent programs (Sutter and Larus, 2005).

In the next section we give examples of some insidious problems that can arise in concurrent programs. Throughout the next section we refer to these problems as *concurrency hazards*, or simply hazards (Goetz et al., 2006).

5.2 Concurrency Hazards

Unpredictability along with inadequate synchronization might cause the interleaving of threads to yield undesirable results. A concurrency hazard that stems from insufficient synchronization is a *race condition* (Netzer and Miller, 1992). Race conditions happen when the correctness of a computation depends on the timing or interleaving of the threads performing the computation (Goetz et al., 2006; Stallings, 2011). For example, a race condition might happen when two threads try to modify shared data or when a thread is modifying data and the other is trying to read the same data (Subramaniam, 2011). Therefore, depending on which thread accesses the shared data first, the result can be different (Gatlin, 2004).

Listing 5.1 illustrates a simple example of Java code that when accessed by multiple threads might result in a race condition. `UnsafeSequence` is supposed to generate and return a unique integer value each time `getNext` is invoked. However, depending on how the threads interleave, it may yield erroneous results. As pointed out by Goetz et al. (2006), the problem is that two threads could call `getNext` and receive the same value. Figure 5.2 illustres an interleaving that can lead to this problem. As shown in Figure 5.2, although `value++` appears to be a single instruction (i.e., indivisible operation), in fact, when compiled down to Java bytecodes such an instruction becomes three instructions: fetch the current value of the variable, add one to this value, and write the new value to the variable. During execution, if two threads read `numValue` almost at the same time, both see the same value and add one to it.

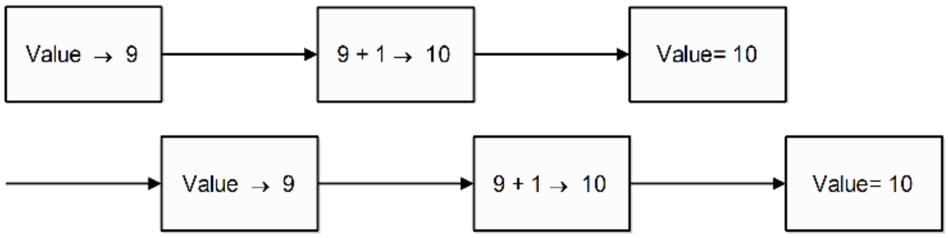


Figure 5.2: An interleaving that leads to a race condition. In this figure, time runs from left to right, and each line represents the instructions executed by each thread. This figure was adapted from Goetz et al. (2006).

Listing 5.1: Non-thread-safe sequence generator; this piece of code is taken from Goetz et al. (2006).

```
public class UnsafeSequence {
  private int numValue;

  public int getNext() {
     return numValue++;
  }
}
```

Another concurrency hazard that might arise when threads compete for resources is a *deadlock*. More specifically, a deadlock is an impasse that happens when two or more threads are waiting for shared resources that are owned by each other (Coffman et al., 1971; Stallings, 2011). Given that locks are dispersed throughout multithreaded programs, these programs are prone to deadlocks (Gatlin, 2004).

Given two threads T_1 and T_2. T_1 owns lock L_1 while T_2 holds L_2. If T_1 attempts to acquire L_2 without releasing L_1 and T_2 tries to acquire L_1 without releasing L_2, both threads will freeze indefinitely. In this situation, none of the threads can resume execution nor release their resources (Tanenbaum, 2007). An example of how this can happen is shown in Listing 5.2. The `leftRight` and `rightLeft` methods try to acquire the `left` and `right` locks. Problem arises when one thread calls `leftRight` and another calls `rightLeft` and the execution of these threads interleaves as shown in Figure 5.3. According to Goetz et al. (2006), this sort of deadlock can be referred to as *lock-ordering deadlock*.

Another concurrency hazard similar to a deadlock, is a *livelock*. A livelock is a situation in which two or more threads keep changing their states in response to changes in the other threads (Stallings, 2011). This situation is similar to a deadlock because no progress is made by the involved threads, however, the difference is that neither thread is blocked nor waiting for shared resources.

5.3 Concurrent Testing Techniques

As stated by Al-Iadan (2001), Chen and MacDonald (2007), and Eytani et al. (2007), there are basically two main techniques for revealing concurrency-related faults: *static* and *dynamic* techniques. The next sections outline the characteristics of these techniques and give a brief summary of the related literature.

Listing 5.2: Example of lock-ordering deadlock; this piece of code is taken from Goetz et al. (2006).

```java
public class LeftRightDeadlock {
  private final Object left  = new Object();
  private final Object right = new Object();

  private void leftRight() {
    synchronized (left) {
      synchronized (right) {
        doSomething();
      }
    }
  }

  private void rightLeft() {
    synchronized (right) {
      synchronized (left) {
        doSomething();
      }
    }
  }
}
```

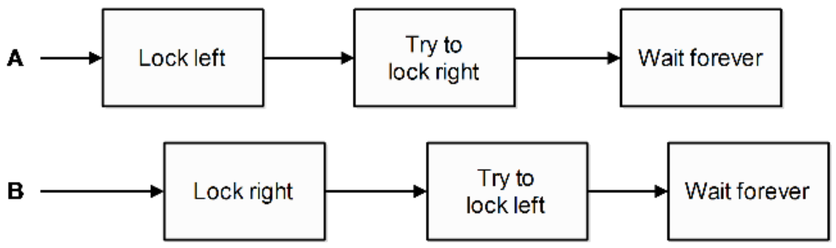


Figure 5.3: Example of lock-acquisition order that leads to a deadlock. This figure was adapted from Goetz et al. (2006).

5.3.1 Static Techniques

Static techniques inspect the code of programs without running it (Al-Iadan, 2001). These techniques circumvent non-determinism by applying different static methods to reveal potential concurrency hazards. Static techniques are able to identify many concurrency hazards, but emphasis is given to deadlocks and data races. Since static techniques do not entail code execution, they are suited to programs containing hard-to-reach code (Raza, 2006). However, these techniques fall short of detecting hazards that call for feasibility analysis (Netzer and Miller, 1992; Chen and MacDonald, 2007). Since static techniques have to deal with NP-Complete and undecidable problems, many assumptions are made during program analysis (e.g., it is assumed that all execution paths in a program are possible). As a result, static techniques are known to yield false positives and false negatives. Another drawback is that they do not take different inputs into account. Next subsections outline some static techniques.

5.3.1.1 Model Checking

Model checking (Clarke et al., 2009; Ben-Ari, 2010) is a static technique that has been proven useful for the verification of concurrent programs. Briefly, given a finite-state model and properties of the program under consideration, model checking traverses all states in a brute-force fashion to check whether the given properties hold for that model (Baier and Katoen, 2008). Typically, the properties validated through model checking are qualitative (e.g., Does the program ever reach a deadlock situation?). In this context, properties are expressed as formulas in temporal logic or invariants (predicates) (Eytani et al., 2007).

The main problem that model-checking algorithms have to cope with is the *state space explosion* (Clarke et al., 2009). The state space of concurrent programs grows exponentially with the number of threads (Malkis et al., 2007), thereby the number of states in a concurrent program with many threads can be very large. It follows that model checking takes a long time and requires high computational power. Thus, a major research thrust is focused on creating effective data structures and algorithms able to deal with large search spaces within a reasonable amount of time (Clarke and Wing, 1996). Much of this knowledge has been translated into state-of-the art tools.

One of the leading model-checkers is SPIN (SImple PROMELA INterpreter) (Holzmann, 1997), which was written in C and supports the verification of concurrent and distributed programs. Models in SPIN are written in PROMELA (PROcess or PROtocol MEta LAnguage), which is a constrained yet efficient language. The syntax and semantics of expressions and assignment statements in PROMELA are similar to those in C. The basic data types are integers (whose sizes range from one to 32 bits), booleans, and one-dimensional arrays. The other features of the language are used to represent concurrent constructs and related concepts, e.g., processes, message channels, and atomic statements. To speed up the verification of PROMELA models, SPIN generates an optimized model checking program in C for each model and prescribed property to be verified (Ben-Ari, 2010).

Manually generating models in languages as PROMELA is error-prone (Eytani et al., 2007). In an attempt to circumvent this step, model-checkers that semi-automatically or automatically generate models from programs written in HLL have been developed. A notable example is Java PathFinder (Havelund and Pressburger, 2000). Java PathFinder verifies programs written in Java, as opposed to SPIN that uses a modeling language. As a result, the verification is more realistic (Ben-Ari, 2010), however, the size of the models that can be taken into account is limited because Java programs are complex abstractions with many states. Other contributions in this area include Bandera (Corbett et al., 2000) and SLAM (Ball et al., 2001).

Being a systematic and exhaustive technique makes model checking less scalable. A more scalable static technique concentrates on pruning away non-determinism from HLL, making them more amenable to conventional testing techniques. These programming languages are presented in the next subsection.

5.3.1.2 Deterministic HLLs

Several *deterministic programming languages* have been developed (Rinard and Lam, 1998; Thies et al., 2002; Bocchino et al., 2009). Programs written in these languages are always deterministic. Jade, which was designed and implemented by Rinard and Lam (1998), is an imperative HLL that uses programmer-specified annotations to remove concurrency while preserving the semantics of sequential code. Thies et al. (2002) developed StreamIt: a language for streaming computations that ensures deterministic behavior by allowing inter-process communication to occur only via FIFOs.[2] Bocchino et al. (2009) augmented the Java language with an effect system (Nielson and Nielson, 1999) that enforces deterministic semantics via compile-time type checking.

5.3.2 Dynamic Techniques

In contrast with static techniques, dynamic techniques investigate run time information (Al-Iadan, 2001). In general, these techniques capture run time information by statically or dynamically instrumenting programs. This instrumentation incurs high computational cost, but an advantage of dynamic techniques is that only the executed paths are examined. However, as the size of a program grows, so does the number of feasible paths. Next subsections describe dynamic techniques, special emphasis is given to tools.

5.3.2.1 Noise Makers

Traditional testing is not well-suited for concurrent programs as it fails to cover many interleavings (Ball et al., 2011). When conventional testing is carried out with no additional strategy to tamper with execution, only minor variations of the same thread interleaving tend to be exercised (Wang et al., 2011). Aimed at exploring an increasing number of interleavings during testing, researchers and practitioners often turn to *noise makers*. A

[2] FIFO (also known as named pipe) is an inter-process communication mechanism used on Unix-like systems.

noise maker is a software testing tool that makes each execution of a concurrent program to behave slightly different by seeding scheduling noise. Typically, these tools rely on instrumenting programs with conditional synchronization primitives (e.g., `yield`, `sleep`, and `wait`) to induce the execution of different interleavings (Eytani and Latvala, 2007). Some noise makers also implement heuristics to decide (randomly or based on some statistics) whether the seeded primitives should execute.

By algorithmically exploring the space of interleavings through the introduction of scheduling noise, noise makers increase the amount of interleavings covered by each test (Eytani et al., 2007). Thus, tests are more likely to uncover error-yielding interleavings, improving the efficiency of testing. For this reason, a number of noise makers have been developed (Edelstein et al., 2002; Stoller, 2002; Eytani et al., 2003).

A notable example is the testing framework Chess (Ball et al., 2011), which includes a noise maker for C# programs. By instrumenting preemptions at synchronization points, Chess progressively explores the space of interleavings aimed at finding failure-manifesting schedules. Although the number of preemption combinations is exponential, Chess employs heuristics to cope with the state-space explosion problem. Two tools similar to Chess are CalFuzzer (Joshi et al., 2009) and CTrigger (Park et al., 2009). Likewise, both noise makers instrument programs with preemptions to reveal erroneous schedules.

As noted by Eytani and Latvala (2007), a downside of this technology is that some noise makers tend to seed too many synchronization primitives, making debugging activities more difficult. This led researchers to look into strategies that indicate where scheduling noise should be introduced (Ben-Asher et al., 2006).

5.3.2.2 Race Detection

A substantial amount of theoretical work has been carried out in the area of race detection (Adve et al., 1991; Netzer and Miller, 1991; Raza, 2006; Hafeez et al., 2012) and many tools have been developed. Early implementations had severe limitations: some were unable to examine programs containing more than one semaphore (Lu et al., 1993). Nevertheless, the increasing interest in concurrency has prompted the development of more sophisticated tools. As noted by Raza (2006), current tools detect races by keeping track of either the order in which threads access shared data or the lock acquisition sequence.

Techniques for detecting race conditions can be based on static analysis, dynamic analysis, or a combination of both. Thus, throughout this subsection, instead of limiting our discussion to the type of analysis, race detection tools are further classified into one of three categories: *on-the-fly*, *ahead-of-time*, and *post-mortem*. Each of these categories has its own advantages and disadvantages. By nature, on-the-fly approaches tend to use dynamic analysis, whereas ahead-of-time approaches examine static artifacts as source code. Post-mortem approaches combine static and dynamic analysis (Raza, 2006).

On-the-fly race detection techniques suffer from the same drawbacks as dynamic techniques. The main disadvantage is that they impose a high computational overhead. Another issue is that the non-deterministic nature of schedulers further complicates the detection of race conditions by on-the-fly techniques. Finally, on-the-fly tools are not able to examine all parts of programs, only executed paths are taken into account (Raza, 2006).

An example of on-the-fly race detector is Eraser (Savage et al., 1997). Such a tool instruments binary programs to log every access to shared memory. The instrumentation checks whether shared memory accesses are performed only after proper lock acquisition operations. The central idea of Eraser is the Lockset algorithm. At run time, Eraser infers which locks protect each shared variable, forming a lockset for each variable. As the program runs, the candidate lockset for each variable is updated with the locks held by every thread that accesses the shared variable in question.

For instance, when a given shared variable V is initialized, its candidate lockset includes all possible locks. Whenever V is accessed, its set of locks is updated with the intersection of its lockset and the accessing thread's set of locks. That is, two locks, L_1 and L_2, are in the lockset of a given shared variable V if all threads that accessed V were holding L_1 and L_2. This process is named lockset refinement, and assures that locks that protect V are in V's lockset. Figure 5.4 shows how potential data races are detected using the lockset refinement. The left column represents the program instructions, which are executed from top to bottom. The middle column represents the set of locks held by the running thread and the right column is the lockset of the variable V after the execution of each statement. Considering that this example has two locks, the lockset is initialized with both locks, i.e., L_1 and L_2 (as illustrated in ❶). Whenever V is accessed, its lockset is refined to contain the set of locks held by the accessing thread. As shown in ❷, the lockset is updated as follows: `Locks Held` $\cap$ `Lockset` $= \{L_1\} \cap \{L_1, L_2\} = \{L_1\}$. Afterwards, V is accessed again, but this time only L_2 is held by the running thread. Thus, the intersection of the two sets is empty (as highlighted in ❸), which indicates that no lock protects V for the entire program. Savage et al. extended the lockset algorithm to deal with some special cases and avoid issuing warnings when the resulting lockset is empty but the program is free from data races, e.g., single-writer, multiple-reader locks. Subsequently, Choi et al. (2002) developed a tool similar to Eraser for Java programs. Choi et al. improved upon Eraser in that their tool incorporates static analysis to avoid unnecessary investigations at run time.

The main shortcoming of Eraser is that it is targeted at mutex synchronization operations, so it is unable to work properly when other synchronization primitives are used on top of mutexes. Additionally, since it relies on instrumentation, modified programs show some slowdown.

Many ahead-of-time techniques are based on strong-type checking. Boyapati and Rinard (2001), for example, devised a static type system for multithreaded Java programs. Their type system enables programmers to specify the locking discipline of programs. According to them, their type system ensures that any well-typed program is free from data races.

Lockset Algorithm

Program	Locks Held	Lockset	
↓	{}	{L_1, L_2}	❶
lock(L_1);			
↓	{L_1}		
V += 1;			
↓		{L_1}	❷
unlock(L_1);			
↓	{}		
lock(L_2);			
↓	{L_2}		
V += 1;			
↓		{}	❸
unlock(L_2);			
↓	{}		

Figure 5.4: Lockset refinement: when each access to a variable is protected by a lock, no lock protects the underlying variable for the whole program. During execution, when the accessing thread's set and the lockset are disjoint, Eraser issues a warning indicating that no lock protects V. This figure was adapted from Savage et al. (1997).

Flanagan and Freund (2000) also developed an ahead-of-time tool for detecting race conditions. Their tool, named rccjava, implements a type system whose goal is to check if the locks that protect a given variable are acquired every time the variable is accessed. The tool requires programmer-supplied information to verify the locking strategy. This additional type information is declared in Java comments. Later on, Flanagan and Freund (2001) improved rccjava, enabling it to be used on large, real-world programs.

Similarly to on-the-fly techniques, post-mortem techniques instrument programs to record run time information. However, the analysis of the log is mostly done post-execution. While the instrumentation overhead incurred by these techniques is lower than on-the-fly techniques, the resulting log files can be very large. Given that, researchers strive to strike a balance between the amount of information captured at run time and the precision and size of log files. Due to the fact that a great deal of the analysis performed by post-mortem techniques relies on logs, these techniques suffer from the same limitations as on-the-fly techniques: the analysis is limited to the executed paths (Raza, 2006). Most record and playback tools, which are discussed in the next subsection, can be regarded as post-mortem techniques.

5.3.2.3 Record and Playback

The non-determinism of concurrent programs has a number of important implications, including some that pose challenges to testing and debugging. First, after finding an error-manifesting interleaving, it is difficult to precisely reproduce such interleaving. Differently from deterministic programs, re-executing a concurrent program does not guarantee that the erroneous behavior will happen again. The concurrency hazards that cannot be reproduced with high probability because they occur only under some interleavings are called Heisenbugs (Grötker et al., 2012). Second, when the interleaving can be easily reproduced, trying to analyze the problem using a debugger or print statements may tamper with the execution (i.e., probe effect), causing the erroneous behavior to disappear (McDowell and Helmbold, 1989). In essence, both problems can be ascribed to the lack of control over which interleavings execute each time a program runs (Ball et al., 2011). Such a lack of repeatability stems mostly from the fact that the scheduler cannot be directly controlled.

To facilitate testing and debugging, researchers have looked at ways to replay executions. Typically, replay mechanisms have two phases: *record* and *playback* (Eytani et al., 2007). During the record phase, information concerning the scheduling is recorded, e.g., order in which shared variables are accessed and synchronization events within the threads. In the playback phase, sometimes referred to as *replay* phase, the information captured in the record phrase is used by the replay mechanism to ensure that executions cover the same thread schedule.

As Eytani et al. (2007) remarks, depending on the degree of control that the replay engine has over the execution environment, replay mechanisms can be divided into two groups: *full replay* and *partial replay.* Implementing full replay is complex and entails recording a large amount of information. Partial replay, which makes programs execute as if the scheduler were deterministic, is easier to implement and enough to support testing and debugging.

DEJAVU (Deterministic Java Replay Utility) (Choi et al., 2001), which was implemented as part of an early[3] version of Jikes RVM, provides full replay. DEJAVU captures wall-clock time[4] values during execution in record mode. These recorded values are used

[3]Jikes RVM was formerly known as Jalapeño.

[4]Wall-clock time (or wall time) is the time taken by a computer to completely execute a program (Raymond, 1996). Instead of taking into account only the time the processor spends executing the program, wall-clock time also factors in the time spent in programmed delays or waiting for resources to become available.

during replay to deterministically reproduce timed thread events that depend on wall-clock values such as timed waits. Differently from some partial replay mechanisms, DEJAVU deterministically replays the entire execution, not only the order in which synchronization events take place. When it replays a program up to a synchronization primitive (e.g., `monitorenter`), it also replays the whole program and execution environment states. Thus, DEJAVU keeps track of a large amount of information, including the state of each thread (e.g., locks currently owned by each thread) and the dispatch queue. DEJAVU also deals with another source of non-determinism: random values. In record mode, random values read from external devices are recorded and then, in replay mode, these values are replayed deterministically.

To implement full replay in an HLL VM, Choi et al. (2001) took advantage of the fact that early incarnations of Jikes RVM used to rely on a *green thread model*. In a green thread model, the scheduler is part of the execution environment, so all thread management decisions are made by the HLL VM. Currently, only HLL VMs tailored to resource-constrained devices use green threads (Joisha et al., 2002; Simon et al., 2006; Aslam et al., 2008). The reason behind such a design decision is rooted in the architecture of resource-constrained devices. In a green thread model all threads are run atop of a single native thread. Thus, this model is a natural fit for the uniprocessor and single core architecture of most resource-constrained devices. Modern HLL VMs use native threads to take advantage of multiprocessors (Jikes RVM Project, 2013; Oracle Corporation, 2013b; Wimmer et al., 2013), which delegates the management of threads to the OS and makes the implementation of replay capabilities more difficult. As a result, to cope with the amount of information that has to be recorded in the many software layers involved, researchers have been implementing full replay at the OS level or in hardware. Most proposals hinge on instrumentation or special hardware support.

The OS-based technique of Russinovich and Cogswell (1996) captures thread switches on a uniprocessor. They modified the Mach OS so that it records each thread switch. Since their technique does not replay the entire state of the OS nor the state of its thread package, their replay engine enforces the desired thread schedule by notifying the scheduler which thread it should execute next at each thread switch.

Holloman's (1989) technique is similar to Russinovich's and Cogswell's. The main difference is that Holloman uses instrumentation to capture scheduling information. Specifically, exception handlers are instrumented into programs. These handlers capture all exceptions, including the ones related to scheduling, which are sent from the Unix OS to processes.

Olszewski et al. (2009) implemented Kendo, which is a deterministic, partial replay system for race-free programs. Kendo achieves deterministic replay by logging the sequence of synchronization operations in record mode. In replay mode, Kendo assigns deterministic identifiers for each thread and increases the logical clock of each thread deterministically. Whenever a thread acquires a lock, its logical clock is incremented, which ensures that locks are acquired in the same order as they were recorded. Kendo's main shortcoming is that it requires special hardware (i.e., deterministic performance counters) that is not available on some platforms (Weaver and McKee, 2008).

Similarly to Choi et al. (2001), Bergan et al. (2010) tackle the non-determinism problem in a platform-centric manner. Bergan et al. designed and implemented CoreDet, which comprises a compiler and a runtime system for replaying C/C++ multithreaded programs. A program compiled with their infrastructure always executes deterministically, yielding the same result even in the presence of data races. Bergan et al. investigated two basic ways to enforce determinism. The first one is based on tracking data ownership and serializing execution whenever inter-thread communication is detected. The second is a buffering approach based on versioned memory. A deterministic commit protocol is used to make threads aware of any access to shared data. Results suggest that the second approach tends to scale better, serializing execution less often. Calvin (Hower et al., 2011), which is a hardware-based approach, uses a replay algorithm similar to CoreDet's.

Dthreads (Liu et al., 2011) is a replacement for the POSIX threads application programming interface (API) (Butenhof, 1996), also known as pthreads, that provides deterministic multithreading for C/C++ programs. Dthreads supports all synchronization primitives implemented by POSIX threads. According to Liu et al., Dthreads can even outperform pthreads in some cases.

5.4 Mutation Testing for Concurrent Programs

As stated in Chapter 3, mutation testing has been used as a gold standard to gauge the effectiveness of other testing techniques. Given that, it is understandable that some researchers have been trying to apply mutation testing to concurrent programs. However, besides having to cope with non-determinism, applying mutation testing entails an extra step: devising mutation operators. Most conventional operators fall short of directly mutating constructs associated with concurrency and synchronization. Hence, several sets of mutation operators that mimic concurrency-related faults have been proposed.

Delamaro et al. (2001) proposed a set of 15 mutation operators for concurrent Java programs. Operators in this set are grouped in four categories: *(i)* operators that modify locks, *(ii)* change methods related to wait set manipulation (e.g., `wait` and `notify`), *(iii)* change the invocation of synchronized methods, and *(iv)* modify methods that implement thread collaboration (e.g., `sleep`, `yield`, and `join`). Later, Bradbury et al. (2006b) created a set of 24 mutation operators for concurrent Java programs. Differently from the operators created by Delamaro et al., these operators mutate the most recent concurrency constructs in Java 5 and some built-in concurrent data structures, e.g., barriers, latches, hash maps, queues, and thread pools. To design operators whose resulting mutants are representative of real faults, Bradbury et al. based their operators on a taxonomy of concurrent fault patterns (Farchi et al., 2003). The mutation operators proposed by Bradbury et al. are divided into five categories: *(i)* operators that modify parameters of concurrent methods, *(ii)* change concurrency-related method calls by removing or replacing them, *(iii)* modify keywords by adding or deleting them, *(iv)* switch concurrent objects, and *(v)* make changes to critical regions by shifting, expanding, shrinking, or splitting them.

Silva et al. (2012) came up with a set of mutation operators for MPI (Message Passing Interface) programs. MPI is a message-passing API designed to support the implementation of concurrent programs written in C or Fortran. According to Silva et al., their set of mutation operators reflects typical mistakes that programmers make when implementing concurrent programs. Silva et al.'s operators are based on the literature, mainly on the fault taxonomy proposed by DeSouza et al. (2005). The operators are divided into three groups, namely, *collective*, *point-to-point*, and *all*. Collective contains the operators that are applied to collective-communication functions. Point-to-point includes operators concerned with modifying point-to-point communication functions, this group is further split into three subgroups: operators applied to send functions, operators that change receive functions, and operators that modify other point-to-point functions. The last group comprises operators that can be applied both to collective and point-to-point functions. These operators were implemented in a tool called ValiMPI_Mut (Silva, 2013).

Gligoric et al. (2013b) investigated selective mutation for concurrent mutation operators. The mutation operators used in the study are the ones devised by Bradbury et al. and three new operators proposed by Gligoric et al. themselves. Their results indicate that operator-based selection is better than random mutant selection. More importantly, their results would seem to suggest that sequential and concurrent mutation operators are independent, which reflects the importance of examining concurrent mutation operators. According to Gligoric et al., selective mutation for concurrent code yields lower savings than for sequential code and selecting operators based on either the number of mutants or categories of operators does not seem to be effective. Differently from previous studies that provided one selective set of mutation operators, Gligoric et al. provide several sets of concurrent mutation operators so that one can choose a selective set that yields marked savings but with reduced effectiveness or a set that is highly effective, but costly.

Of the mutation tools in Table 3.1, only Paraμ (Madiraju and Namin, 2011), ExMAn (Bradbury et al., 2006a), and MuTMuT (Gligoric et al., 2013a) implement concurrent mutation operators. Given that Paraμ generates mutants through bytecode instrumentation, it implements only operators that entail straightforward changes. The three operators are listed in Table 5.1. As future work, Madiraju and Namin intend to implement Bradbury et al.'s entire set of mutation operators. ExMAn implements more operators than Paraμ. More precisely, as reported by Bradbury et al. (2007), ExMAn implements a subset of the operators that Bradbury et al. themselves came up with in previous work (Bradbury et al., 2006b). The concurrent mutation operators implemented in ExMAn are shown in Table 5.2. MuTMuT implements all mutation operators defined by Bradbury et al. and three new operators proposed by Gligoric et al. (2013b), which totals 27 concurrent mutation operators. Instead of implementing a mutation generation engine from scratch, the authors of MuTMuT extended Javalanche to mutate programs. Initially, Javalanche did not support any concurrent mutation operators. According to Gligoric et al., the authors of Javalanche included their extension in the publicly available Javalanche distribution.

Table 5.1: The concurrent mutation operators implemented in Paraμ.

Operator Acronym	**Description**
`ASTK`	Add `static` keyword to method.
`RSTK`	Remove `static` keyword from method.
`RSK`	Remove `synchronized` keyword from method.

However, the main drawback of Paraμ, ExMAn, and MuTMuT is that they do not allow for deterministically re-executing mutants. Applying mutation testing to deterministic programs is rather straightforward in that the correctness of a given output can be easily determined: it comes down to comparing the output under consideration with the output of the original program using the same test input. Nevertheless, the same does not hold for concurrent programs because of their non-deterministic nature. Given a certain input, a concurrent mutant is killed if the set of outputs of this mutant for all possible schedules differs from the set of outputs of the original program for all possible schedules. Thus, the cost of applying mutation testing to concurrent programs is even greater in comparison to sequential programs: apart from having to execute the test inputs on many mutants, each test input has to be executed for multiple possible thread schedules. This, taken together with the fact that even a small program with a few dozen lines of code and a couple of threads can result in a large number of schedules, makes mutation testing unwieldy for concurrent programs.

Therefore, without the possibility of enforcing deterministic re-executions, the notion of correctness has to be reconsidered. In an attempt to circumvent this limitation, Offutt et al. (1996b) suggested a modified definition of correctness for concurrent programs. The proposed approximation consists in executing the original program n times with the same input to create n outputs o_i $1 \leq i \leq n$. The resulting set, $\Omega = \{o_1, o_2, \ldots, o_n\}$, is an approximation of the feasible output set. Using such an approximation, a mutant is considered dead whenever its output (o_m) does not appear in Ω (i.e., $o_m \notin \Omega$). Naturally, how well Ω represents the true feasible output set of the program hinges on the value of n. This approximation is a good idea provided that it is possible to enforce a considerable amount of distinct executions, each exploring a different synchronization sequence. Nevertheless, when a concurrent program is run with no additional strategy to tamper with its execution, only slight variations of the same synchronization sequence tend to be executed repeatedly (Wang et al., 2011).

Table 5.2: The concurrent mutation operators implemented in ExMAn. This table was taken from Bradbury et al. (2007).

Operator Acronym	**Description**
`MXT`	Modify method-X time (`wait`, `sleep`, and `join`).
`MSP`	Modify synchronized block parameter.
`RTXC`	Remove thread method-x call (`wait`, `sleep`, `join`, `yield`, `notify`, and `notifyAll`).
`RNA`	Replace `notifyAll` with `notify`.
`RJS`	Replace `join` with `sleep`.
`ASTK`	Add `static` keyword to method.
`RSTK`	Remove `static` keyword from method.
`ASK`	Add `synchronized` keyword to method that contains a synchronized block.
`RSK`	Remove `synchronized` keyword from method.
`RSB`	Remove synchronized block.
`RVK`	Remove `volatile` keyword.
`SHCR`	Shift critical region (up and down).
`SKCR`	Shrink critical region.
`EXCR`	Expand critical region.
`SPCR`	Split critical region.

5.5 Concluding Remarks

In this chapter some key concepts related to concurrency were discussed. Initially, the discussion focused on the implications that the non-determinism of concurrent programs have for testing and debugging. As mentioned, the problem is twofold. Firstly, it turns out that traditional testing fails to cover many interleavings. Secondly, due to the unpredictability of concurrent programs, re-executing these programs does not always yield the same output, which complicates testing and debugging.

Despite the efforts of language developers to create more concurrency-friendly HLLs, reasoning about, implementing, testing, and debugging concurrent programs are still rather complex activities. To keep the collaboration among threads in check, several abstractions have been devised and included into programming languages as, for example, semaphores and monitors. However, even using these basic synchronization constructs, implementing concurrent programs is error-prone. Two common concurrency-related hazards are deadlocks and race conditions, which were briefly described in this chapter.

To provide background for the next chapter, we outlined some of the techniques to uncover and reproduce concurrency-related hazards. The presentation of these techniques was divided into two main sections, the first described static techniques and the second covered dynamic techniques. In these sections, particular emphasis was given to the description of tools. Some dynamic techniques were sub-classified into three subgroups, one of which is post-mortem. Record and replay tools fall into the post-mortem subgroup. In addition, some efforts to apply mutation testing to concurrent programs were discussed. These studies indicate that there has been some interest in overcoming the problems that stem from non-determinism in order to make mutation testing more practical. Next chapter presents our record-and-playback implementation, which is built around Java's built-in locking mechanism.

Chapter

6

Recording and Replaying Multithreaded Java Programs and Progressively Exploring Interleavings

Many software practitioners have been exposed to concurrent programming in order to reap the performance benefits of multicore computers. This has brought concurrent HLLs, such as Java, to the fore. However, as mentioned in the previous chapter, the potential benefits provided by concurrency do not come without costs: writing concurrent programs is error-prone and testing and debugging such programs is notoriously complex.

Concurrent programs are hard to test and debug mainly because of their inherent nondeterministic behavior. No mainstream HLL provides a way to deterministically control which schedules will take place each time a program executes. During execution, thread-management decisions are made by the underlying scheduler, which makes it difficult to test whether a program executes correctly on every possible thread schedule.

In general, when conventional testing is employed without any additional strategy to either control or tamper with execution, only minor variations of the same thread interleaving tend to be exercised (Wang et al., 2011). Thus, many error-inducing thread interleavings manifest themselves only when the programs are already in production. In fact, even a common practice such as stress testing is not effective to uncover these hard-to-diagnose problems. Simply put, executing the same test multiple times does not ensure that an erroneous interleaving will turn up.

Record and playback tools are an approach to cope with the unpredictability of concurrent programs during testing and debugging. These tools make it possible to capture information about a given execution and then, in a later stage, use the acquired information to deterministically re-run the previous execution. Due to the technology-centered nature of this problem, most realizations of the aforementioned approach deal with it in an implementation-based way. Taking Java as an example, which uses an HLL VM, several approaches can be used to enforce a certain program `P` to be deterministically re-executed. Some possible approaches are the following:

(i) Modifying the HLL VM thread scheduler in such a way that the scheduling of threads follows a pre-established thread schedule;

(ii) Usually, modern HLL VMs provide APIs that make it possible for end-users to access internal runtime information, e.g., the Java Platform Debugger Architecture (JPDA).[1] By accessing the running state of an HLL VM through its debugging API, for instance, it is possible to monitor the thread scheduling sequence during the first execution of `P`. The gathered scheduling information can then be used to force the preceding scheduling sequence in later executions.

(iii) Instrumenting `P` in an ahead-of-time fashion so that it keeps an encoding of the schedules explored during the first execution. Afterwards, `P` can be re-instrumented in order to enforce the previous scheduling sequence.

Each approach has its advantages and limitations. A significant limitation of the first approach is that mainstream HLL VMs no longer use non-native threads (i.e., green threads). In other words, their threading model does not manage threads through internal schedulers. Rather, the implementation of threads in these HLL VMs relies on native OS capabilities. Most JVM implementations that are tuned for performance implement threads natively, e.g., Maxine VM, Jikes RVM, and HotSpot. Typically, the green-thread model has been employed by HLL VMs tailored to low-end devices, which are a mixture of OS and HLL VM, e.g., Squawk.

As for the second approach, it could be implemented by piggybacking on the JPDA API. Considering such an implementation, the monitoring program can set breakpoints in certain locations of `P` to record the synchronization sequence. Upon re-executing `P`, the monitoring program uses the breakpoints to exert control over the execution. The

[1] `http://docs.oracle.com/javase/6/docs/technotes/guides/jpda/`.

monitoring program acts as a specialized scheduler that is triggered when breakpoints are reached, taking over the execution by pausing and resuming threads according to the pre-recorded synchronization sequence. However, a potential limitation of this approach is that breakpoints are likely to incur in substantial overhead; resulting in re-executions that are much slower than the original execution. Another limitation is that the number of breakpoints required to enforce re-executions can quickly become unwieldy (Delamaro, 2004).

The third approach basically entails transforming the original program so that it captures the synchronization sequence that took place during its execution. Additionally, before subsequent executions, the original program is transformed to enforce the pre-recorded synchronization sequence. Essentially, the main practical limitation of this approach is the overhead caused by the instrumentation code.

The main advantage of the first approach is its simplicity. In such an approach all changes are local to the scheduler. As a workaround to the lack of a scheduler, the second and the third approach use alternative ways to overcome this limitation. The performance of these alternative solutions is, however, subpar in comparison to the first approach.

In this chapter, we present how we retrofitted record-and-playback and interleaving exploration capabilities into Maxine VM. Given that Maxine VM does not have an internal scheduler, the approach we adopted to implement the aforementioned features relies on run-time code transformations and builds on the pre-processor and bytecode transformation facilities of the chosen JVM. Also, our HLL VM-based approach is built around Java's built-in locking mechanism, taking advantage of how this mechanism is realized by JVMs and their intermediate language (i.e., Java bytecodes). In view of the fact that our solution is heavily based on how Java implements concurrency through its intermediate language, we surmise that the key elements of our solution can be easily adapted to other JVM implementations that support bytecode transformation at load or run time.

The contributions of our HLL VM-based approach are threefold. First, by using a lightweight record instrumentation, our implementation is able to capture debugging information about the behavior of the running threads, e.g., the order in which threads execute synchronized code and when they enter and exit the waiting set of shared objects. Second, using such an information, our HLL VM-based approach is able to deterministic replay the execution of Java programs. Third, our implementation can automatically and progressively explore new interleavings based on a previously recorded synchronization sequence. Given that synchronization sequences are expressed in a compact and user-friendly textual representation, testers can drive the exploration of new schedules by editing synchronization sequence files.

The remainder of this chapter is organized as follows. Section 6.1 gives background on Java's synchronization mechanism, describing synchronized methods and blocks, which are the basic constructs upon which our implementation builds. Using synchronized regions as checkpoints that can trace and control thread behavior is the central idea of our approach, and is outlined in Section 6.2. Section 6.3 describes how this idea was implemented in MaxineVM. An usage example is presented in Section 6.4. Section 6.5 discusses the interplay between each of the three execution modes in which our implementation can be run and the outcomes produced by each mode. Section 6.6 describes the results of an evaluation of our implementation. Limitations of our implementation are discussed in Section 6.7. We outline related work in Section 6.8 and Section 6.9 presents concluding remarks and future work.

6.1 The Java Synchronization Mechanism: An Overview

In Java, every object has a built-in lock (Sandén, 2004; Arnold et al., 2005a). These locks, used for synchronization purposes, are referred to as *intrinsic locks*, *monitor locks*, or *monitors* (Goetz et al., 2006). Two basic synchronization idioms rely on this built-in locking mechanism: *synchronized methods* and *synchronized blocks*. Listing 6.1 shows an example of synchronized method.

Listing 6.1: Example of synchronized method; code excerpt from The Java Language Specification (Arnold et al., 2005a). Once a thread invokes the method `getBalance` on an instance of `BankAccount`, it has to try to acquire the lock on the instance in question. Successfully acquiring the lock means that the thread can proceed to execute the method. If the thread holding the lock is preempted while executing the method body, other threads trying to invoke `getBalance` will wait until the lock is released.

```
public class BankAccount {
   private long balance;
   ...
   public synchronized long getBalance() {
      return balance;
   }
   ...
}
```

Whenever a synchronized method is invoked, the calling thread attempts to acquire the lock of the underlying object. If the thread successfully acquires the lock, it proceeds to execute the method. Eventually, after executing the method, the lock is released. Another running thread trying to execute that synchronized method will block (i.e., suspend execution) until the lock is released. This means that intrinsic locks in Java are similar to *mutexes* (or mutual exclusion locks), allowing at most one thread to hold the lock (Arnold et al., 2005a; Goetz et al., 2006). Given that only one thread at a time can hold a given lock, it is impossible for two or more invocations of the same synchronized method to interleave. Static methods can also be synchronized. In this case, the executing thread locks on the `java.lang.Class` object associated with the object's class.

As long as a thread holds a lock on a particular object, it can call other synchronized methods on that object (Evans and Verburg, 2012; Javier, 2012). The thread will release the lock when control returns from the outermost synchronized method. Such a reentrant locking scheme keeps threads from stalling in acquiring locks that they already have, allowing recursive method calls and invocations of inherited methods (Arnold et al., 2005a).

Synchronized blocks implement the same mechanism that synchronized methods do. The syntax of a synchronized block is a little bit more verbose, being made up of two parts: *(i)* a reference to the object that will act as the lock and *(ii)* the block of code to be guarded by that lock. Rather than declaring `getBalance` synchronized, as shown in in Listing 6.1, the same behavior can be achieved by turning `getBalance`'s body into a synchronized block as shown in Listing 6.2.

Listing 6.2: Example of synchronized block.

```java
...
public long getBalance() {
   synchronized(this) {
      return balance;
   }
}
...
```

In older JVM implementations, acquiring locks was a costly operation (Niemeyer and Knudsen, 2005). Therefore, short synchronized methods or blocks would result in invocation times significantly larger than the time for actually executing these snippets of code. Currently, Goetz et al. (2006) point out that advances in JVM implementations, mainly with respect to the implementation of the built-in locking mechanism, have made the cost of acquiring locks almost negligible.

Apart from acting as locks, Java objects are also able to act as *intrinsic condition queues* (Goetz et al., 2006). The methods `wait`, `notify`, and `notifyAll` make up the

API for intrinsic condition queues.[2] These methods, implemented in the `Object` class, are fundamental to allow the collaboration between threads.

An object's intrinsic lock and its intrinsic condition queue are closely related. In order to invoke any of the aforementioned methods on a given object, the thread must hold the lock on the object in question. The method `wait` causes the current thread to wait until another thread performs a notification. Shortly after invoking `wait`, the current thread no longer owns the lock and it is added to the object's *wait set*. The wait set is a pool of threads waiting for a notification concerning a certain lock (Smith and Nair, 2005b). The `wait` method should be used as shown in Listing 6.3 (Bloch, 2001; Arnold et al., 2005a).

Listing 6.3: The standard way of using the `wait` method; code excerpt extracted from Arnold et al. (2005a).

```
...
synchronized void doWhenCondition() {
  while (!condition)
    wait();
  //do what must be done when the condition is true...
}
...
```

Notifications are performed by invoking either `notify` or `notifyAll`. Calling `notify` on a given object removes a single waiting thread from the object's wait set.[3] When many threads are waiting on the same object, there is no guarantee regarding the thread that will be chosen to be removed from the wait set. After being removed from the wait set, the chosen thread will not be able to resume its execution until the current thread releases the lock on the underlying object. Further, it is worth stressing that the chosen thread has to compete with other threads that might be trying to acquire the same lock. In other words, upon leaving the wait set, there is no guarantee that the chosen thread will be the one to acquire the lock on the object. Rather than picking only one thread, `notifyAll` removes all waiting threads from the wait set. In general, notification code is similar to Listing 6.4.

At bytecode level, synchronized blocks are supported by two opcodes: `monitorenter` and `monitorexit` (Diehl, 1998; Engel, 1999; Craig, 2005; Lindholm et al., 2012). Lock

[2]Actually, there are three `wait` methods. Throughout this document, only the version that takes no arguments is covered. Arnold et al. (2005a) presents an in-depth description of all versions of `wait`.

[3]On several platforms, there is a third way to leave wait sets: spurious wake-up. A spurious wake-up means that a thread may prematurely leave a wait set without being prompted by a `notify` or `notifyAll` invocation. This happens in platforms in which implementing pthreads is not straightforward (Eckel, 2006, p. 859).

acquisition operations are implemented by `monitorenter`. When a thread executes a `monitorenter` opcode, it tries to acquire the lock for the object pointed to by the reference on the top of the operand stack (Smith and Nair, 2005b). Given that locks operate as counters at bytecode level, once a lock is successfully acquired by a thread, the counter associated with the lock is incremented and the acquiring thread continues execution. When a thread tries to acquire a lock that it already owns, the lock count is incremented and execution continues. Lock-release operations are performed by `monitorexit`. This opcode does that by decrementing the lock count for the object referenced on top of the operand stack. If a lock count becomes zero, then the lock can be acquired by other threads (Smith and Nair, 2005b).

Listing 6.4: The standard way of using the `notify` and `notifyAll` methods; code excerpt extracted from Arnold et al. (2005a).

```
...
synchronized void changeCondition() {
    //change some value used in a condition test...
    notifyAll(); //or notify()
}
...
```

Listing 6.5 illustrates how a synchronized block is compiled to bytecodes. The bytecodes shown in Listing 6.5 are generated from the `getBalance` method shown in Listing 6.2. As can be seen in Listing 6.5, the code that retrieves the value of the variable `balance` is wrapped by `monitorenter` (line 4, ❶) and `monitorexit` (line 8, ❷) opcodes. Moreover, as shown in lines 10 through 14, in case an exception is thrown by the code wrapped by ❶ and ❷, an extra `monitorexit` opcode (line 12, ❸) ensures that the underlying lock is released.

Typically, synchronized methods are not implemented through `monitorenter` and `monitorexit` instructions. Rather, the access flag `ACC_SYNCHRONIZED` (which is an entry in the runtime constant pool) is set for these methods. `ACC_SYNCHRONIZED` indicates that invocations of the method in question must be wrapped in synchronization code. In other words, it is up to the invoking thread to perform the following operations: *(i)* acquiring the lock, *(ii)* invoking the method's body, and *(iii)* releasing the lock whether the method invocation completes normally or abruptly (Lindholm et al., 2012).

Listing 6.5: Bytecodes generated from the `getBalance` method in Listing 6.2. Several attributes as, for instance, the exception table, were omitted from this listing for brevity.

```
ldc_w           //push the Class object corresponding to BankAccount
dup             //duplicate the reference on the top of the stack
astore_1        //store it in local var. 1
monitorenter ❶//lock on the Class object on top of the stack
aload_0         //push the underlying instance of BankAccount
getfield        //retrieve the value of balance
aload_1         //push the Class object corresponding to BankAccount
monitorexit  ❷//release the previously acquired lock
lreturn         //return normally
astore_2        //when an exception is thrown, store it in local var. 2
aload_1         //push the Class object again
monitorexit  ❸//guarantee that the code releases the lock
aload_2         //push the previously thrown exception onto the stack
athrow          //rethrow the exception out of the synchronized block
```

6.2 A Lock-oriented Solution to Record-and-Replay

Our solution capitalizes on the way that Java implements synchronization methods and blocks (i.e., synchronized regions). Essentially, it augments the mutual exclusion mechanisms that wraps these synchronized constructs so that they serve two additional purposes:

(i) logging the order in which threads access synchronized code; and

(ii) controlling the order in which access is granted to synchronized regions.

These two features lay the foundation for deterministically replaying past executions of multithreaded Java programs. With these ancillary features in place, the problem of deterministically re-executing a given program is broken down into two main steps. In the first step, the program is run to record the order in which synchronized regions are accessed. For example, consider Figure 6.1 where two threads (i.e., T_1 and T_2) content for a lock (i.e., L_1) which grants access to a given synchronized region. If during execution these threads interleaved as shown in Figure 6.1(a), the execution log would indicate that T_1 was able to acquire L_1 before T_2. During subsequent executions, the feature described in item *(ii)* uses the information gathered in the previous step to ensure that synchronized regions are accessed in the same order that they were accessed during the first execution. For instance, as shown in Figure 6.1(b), when T_2 reaches the code that tries to acquire L_1 before T_1, our solution enforces the sequence that took place in a previous execution, i.e., Figure 6.1(a). As a result, T_2 is prevented from acquiring the lock, so it has to wait until T_1 is done with L_1.

Put simply, the mechanisms that protect synchronized regions are used as checkpoints by our solution. Upon reaching these checkpoints, instead of simply checking whether the current thread may enter the synchronized region, the actions described in items *(i)* and *(ii)* are also performed.

Our solution differs from full replay mechanisms in that it does not try to enforce a deterministic behavior outside of synchronized blocks. In other words, our solution does not govern the order in which threads reach checkpoints. Instead, it enforces the sequence in which the code wrapped by these checkpoints is executed by the threads. To clarify this point, consider Figure 6.1(b) again, which illustrates that no control is exerted over threads up to the point where they reach synchronized regions.

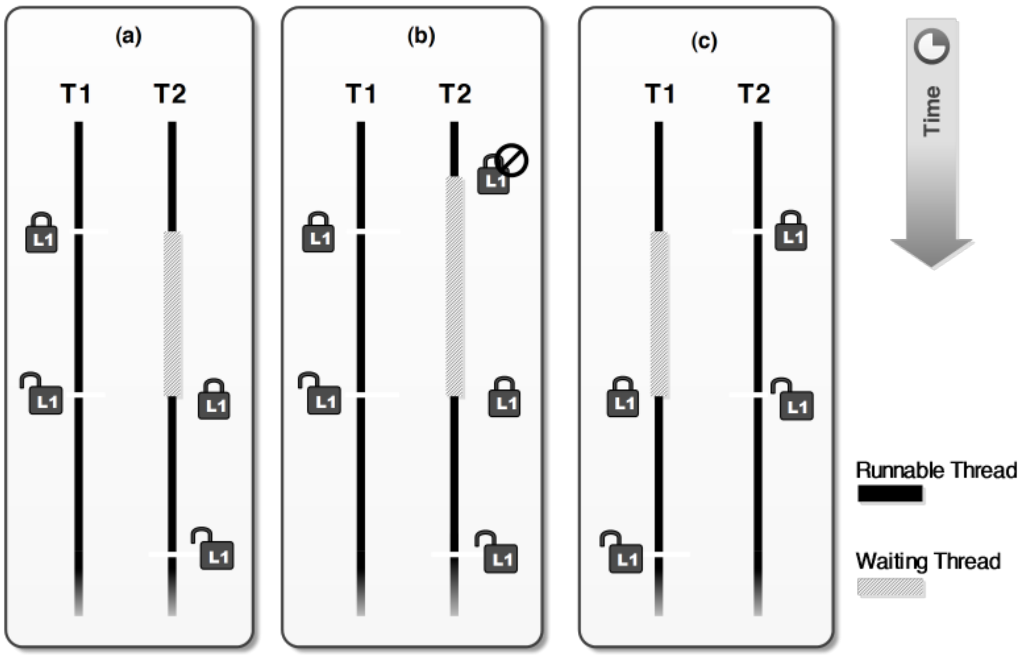


Figure 6.1: Overview of our lock-oriented solution to record and replay.

In addition, our solution capitalizes on the features described in items *(i)* and *(ii)* to explore new interleavings. This exploration is achieved by enforcing that threads run synchronized regions in a different order than the previously recorded execution. In this context, the checkpoint mechanism around locks verifies the order in which the locks were held previously, and tries to explore a new sequence. Given that the interleaving in (a) took place during a first execution, a new interleaving can be explored by ensuring that T_2 acquires L_1 before T_1 as illustrated in Figure 6.1(c). Due to the generic nature of this lock-based approach to record-and-replay, we believe that this idea can be extrapolated to other programming languages whose concurrent constructs are similar to Java. Next section describes how this idea was retrofitted into Maxine VM.

6.3 Proof-of-concept Implementation: Harnessing an HLL VM to Support the Testing of Multithreaded Programs

We extended Maxine VM (Section 2.6) in order to demonstrate the feasibility of retrofitting record-and-playback and interleaving exploration capabilities into an HLL VM. Given that Maxine VM is a native threaded JVM, which means the absence of an internal scheduler, the approach we adopted to implement these features relies on Maxine VM's bytecode pre-processor and bytecode transformation capabilities.

As most post-mortem techniques, our solution breaks down the problem of reproducing a given execution into two phases: record and replay. In our solution, the record phase logs the order in which locks are acquired during execution. The replay phase then uses such an information to ensure that subsequent executions behave in a predictable fashion, following the pre-recorded lock acquisition order. In our proof-of-concept implementation each of these two phases is implemented by an execution mode with the same name.

Both execution modes are geared towards multithreaded Java programs in which non-determinism stems from the sequence in which threads attempt to execute blocks of code guarded by locks. It is assumed that any access to shared data is wrapped in a synchronized block or method, thereby any thread has a deterministic behavior when considered in isolation from others.

In record mode, the program is dynamically transformed at run time. Prior to being executed, each synchronized method or block is transformed in order to monitor lock contention and other synchronization events. In this context, synchronization events are operations that characterize the evolution of threads during execution. An example of synchronization event is lock acquisition. As mentioned in Chapter 5, a sequence of synchronization events is referred to as synchronization sequence. Proceeding from the assumption that every access to shared data is performed inside a synchronized method or block, the goal of executing a program in record mode is to register the order in which synchronization events take place. Given that, a synchronization sequence can be determined by the order in which the threads execute the synchronized code (Delamaro, 2004).

As previously mentioned, unlike synchronized blocks, synchronized methods do not explicitly use `monitorenter` and `monitorexit` instructions, which makes it difficult to trace where the instrumentation code should be inserted. The workings of synchronized blocks make them more convenient for our dynamic instrumentation approach (more on this is discussed later in this section when describing the instrumentation for the replay execution mode). Therefore, prior to inserting the instrumentation code for both execution modes, synchronized methods are rendered into synchronized blocks that span the entire method body. As mentioned, these transformations occur at bytecode level, however, for clarity's sake, in the following listings all instrumentation code is shown as Java source code. Examples of how static and instance methods are modified are presented in Listing 6.6. As can be seen in Listing 6.6, the object acting as lock depends on the type of the method. Instance methods use a reference to the current object (i.e., `this`), whereas static methods use the `Class` object for lock. Methods that already serialize access to their body through synchronized blocks are not modified.

Listing 6.6: Synchronized methods are turned into synchronized blocks. As shown in ❶, instance methods are turned into non-synchronized methods containing synchronized blocks. These blocks span the entire method body. The object upon which the method is invoked is used as lock. Static methods undergo a similar transformation, however, they lock on the `Class` object associated with the object's class (as illustrated in ❷).

```
public class AClass {
 public synchronized void foo() {———→public ↓ ❶ void foo() {
                                       synchronized(this) {
    //method body                        //method body
                                       }
 }                                    }
...                                   ...
 static synchronized void bar() {———→static ↓ ❷ void bar() {
                                       synchronized(AClass.class){
    //method body                         //method body
                                       }
 }                                    }
}
```

After being transformed as shown in Listing 6.6, the resulting methods are further instrumented as follows. The instrumentation for the record execution mode was designed mainly to log when threads acquire locks. Towards this end, the program under test is instrumented by placing probe methods at key locations. First, our implementation inserts opcodes related to a method call before the first line of method bodies (as shown in Listing 6.7, ❷). The call to `afterEnteringSyncBlock` registers that the current thread is the owner of the lock of the object upon which the method was called (i.e., `this`). One of the arguments to `afterEnteringSyncBlock` (i.e., `LockType`) indicates the type of lock that has been acquired (i.e., instance or static). Given that all method bodies were previously wrapped in a synchronized block, this probe logs when the executing thread successfully acquired the underlying lock and is about to run the method in question.

Listing 6.7: Code transformations concerned with the record mode.

```
public void foo() {
   ❶ synchronized(this) {
      ❷ RecordAndPlayback.afterEnteringSyncBlock(this,
            LockType.INSTANCE);
      try {
         //method body
      } finally {
      ❸ RecordAndPlayback.exitingSyncBlock(this,
       LockType.INSTANCE);
      }
   }
}
```

In addition, after inserting the aforementioned method call, the instrumented method body is enclosed within a `try-finally` block. To capture the release of the lock acquired in ❶, our implementation also introduces another method call into the `finally` block (as shown in Listing 6.7, ❸). Such a method call is inserted into the `finally` block because it has to be executed regardless of what happens in the `try` block; whether the block exits normally or abnormally. Note, however, that the method call in ❸ is not necessary for the characterization of the synchronization sequence (Delamaro, 2004).

The replay mode requires a different transformation. Initially, it needs to address the following issue: any thread must check the previously recorded synchronization sequence before trying to acquire a lock and execute the code wrapped in the synchronized block. For example, if a thread T_1 is about to acquire the lock of a given object O_1, there must be a mechanism to check whether this synchronization event (i.e., T_1 acquiring the lock of O_1) is the next one. For a synchronized method, before acquiring the lock means that the instrumentation has to be inserted before the method invocation. However, introducing such instrumentation before every synchronized method call is inefficient, which is one of the reasons for our implementation to transform synchronized methods into synchronized blocks. In doing so, the required instrumentation can be inserted before every synchronized block. Listing 6.8 gives an example of the transformations performed for replaying a synchronization sequence. The method `beforeEnteringSyncBlock` (Listing 6.8, ❶) plays a key role during replay mode, checking whether the current synchronization event (i.e., the current thread trying to acquire the underlying lock) matches the next event in the recorded synchronization sequence.

Apart from checking the synchronization sequence, it is also necessary to remove the first synchronization event from the pre-recorded synchronization sequence whenever a match is found (Listing 6.8, ❷). This must be done inside the synchronized block, assuring that the lock has been acquired before removing the event from the synchronization sequence. Performing such a removal before entering the synchronized block would allow for other threads to lock on the object without respecting the synchronization sequence.

Listing 6.8: Code transformations concerned with the replay mode.

```
public void foo() {
   ❶ RecordAndPlayback.beforeEnteringSyncBlock(this,
         LockType.INSTANCE);
   synchronized(this) {
      try {
      ❷ RecordAndPlayback.nextSyncEvent();
         //method body
      } finally {
      }
   }
}
```

When in record mode, it is important to log every access to synchronized code, thus our implementation also has to deal with wait sets. As stated in Section 6.1, invoking `wait` causes the current thread to enter the wait set of the object upon which the method was called. To register this synchronization event, our implementation has to replace all `wait` invocations by another method whose body includes the following operations:

- an invocation of the timed version of the `wait` method;[4]
- code that logs the fact that the current thread entered the wait set of the object upon which `wait` was invoked.

Although not necessary for the replay mode, registering when threads enter wait sets contributes to debugging by adding information to the recorded synchronization sequence. Similarly, there is no need to replace `notify` and `notifyAll` calls, but doing so makes it possible to log when threads leave wait sets. Thus, our implementation also replaces `notify` and `notifyAll` calls by a method that performs the following operations:

- invokes `notifyAll`;

[4]`http://docs.oracle.com/javase/6/docs/api/java/lang/Object.html`

- adds an event to the synchronization sequence indicating that all threads left the wait set of the object upon which `notifyAll` was called.

As for the replay instrumentation for the `wait` method, it needs to check whether the thread leaving the wait set is supposed to acquire the lock. That is, the instrumentation needs to verify if the lock acquisition by that thread is the next synchronization event.

In summary, every call to `wait` is replaced by a timed-wait and ancillary code to log the related synchronization events. Using a timed-wait prevents the waiting thread from stalling. Registering when threads enter and leave wait sets contributes to add information to the synchronization sequence being recorded.

6.3.1 Progressively Exploring Interleavings

Some frameworks for testing concurrent Java programs provide facilities to repeatedly run the tests with the aim of uncovering fault-manifesting interleavings. Our implementation also allows for exploring different synchronization sequences. Our strategy relies on the textual representation of a valid synchronization sequence captured during record mode. When more than one thread contend for a synchronized block or method, our implementation tries to ensure that each execution takes a different synchronization sequence at a certain point.

Consider two threads, T_1 and T_2, that share a buffer, O. If during record mode T_1 is able to acquire the lock on O, T_1 will read and use the contents of O before T_2. During exploration mode our implementation is able to identify the two possible alternatives, thereby allowing the exploration of the distinct interleaving in a subsequent execution. More precisely, our implementation identifies which threads contend for a certain synchronized block (i.e., shared resource), enumerates the choices available at a certain point in execution, and tries to enforce alternate choices. So, in this case, an event representing "T_1 acquires lock on O" would prompt our implementation to try the alternative: "T_2 acquires lock on O".

Exploring the interplay between threads and synchronized blocks in such a fashion increases the chances of uncovering faults. This also obviates the need for stress testing, which entails increasing the number of threads in hopes of getting more interleaving coverage (Ball et al., 2011). More on how our implementation explores interleavings is detailed in Subsection 6.4.3.

6.4 Example Scenario

This section details the characteristics and usage of our proof-of-concept implementation. The example program used throughout this section is adapted from a Java textbook (Eckel, 2006). Apart from the class containing the `main` method (which is detailed later on), this example has three classes: `WaxOn`, `WaxOff`, and `Car`.

Both `WaxOn` and `WaxOff` interact with `Car`. `WaxOn` implements the `java.lang.Runnable` interface, and instances of this class are responsible for applying wax to `Car` objects. Likewise, `WaxOff` also implements `java.lang.Runnable`, and instances of such a class can be seen as polishing tasks. At run time, instances of `WaxOff` cannot run until an instance of `WaxOn` executes. Similarly, `WaxOn` instances cannot carry on with their execution until a polishing task takes place. `WaxOn`'s implementation is displayed in Listing 6.9.

Listing 6.9: `WaxOn` class.

```
 1  import java.util.concurrent.TimeUnit;
 2
 3  public class WaxOn implements Runnable {
 4    private Car car;
 5
 6    public WaxOn(Car c) {
 7      this.car = c;
 8    }
 9
10    @Override
11    public void run() {
12      try {
13        while (!car.isShiny()) {
14          System.err.println(‘‘Wax on!’’);
15          TimeUnit.MILLISECONDS.sleep(200);
16          car.waxed();
17          car.waitForBuffing();
18        }
19      } catch (InterruptedException e) {
20        System.err.println(‘‘Exiting via interrupt’’);
21      }
22      System.err.println(‘‘Ending Wax On task’’);
23    }
24  }
```

The key method in `WaxOn` is `run`, which implements the waxing process. To simulate the time necessary for waxing, a call to `sleep` is introduced within the `run` method's body (line 15). After waxing, which basically consists in calling `waxed` (line 16), `waitFor-Buffing` is invoked in order to indicate that the waxing is complete. Invoking the method `waitForBuffing` suspends the waxing thread until an instance of `WaxOff` runs (the implementation of `waitForBuffing` is described later).

`WaxOff`'s implementation is very similar to `WaxOn` as shown in Listing 6.10. The main differences are that instead of invoking `waxed` as `WaxOn` does, `WaxOff` calls `buffed` (line 17);

and `waitForBuffing` is replaced by `waitForWaxing`, line 14 (the implementation of `waitForWaxing` is described later).

Listing 6.10: `WaxOff` class.

```java
1  import java.util.concurrent.TimeUnit;
2
3  public class WaxOff implements Runnable {
4    private Car car;
5
6    public WaxOff(Car c) {
7      this.car = c;
8    }
9
10   @Override
11   public void run() {
12     try {
13       while (!car.isShiny()) {
14         car.waitForWaxing();
15         System.err.println(‘‘Wax off!’’);
16         TimeUnit.MILLISECONDS.sleep(200);
17         car.buffed();
18       }
19     } catch (InterruptedException e) {
20       System.err.println(‘‘Exiting via interrupt’’);
21     }
22     System.err.println(‘‘Ending Wax Off task’’);
23   }
24 }
```

As shown in Listing 6.11, in this example, the collaboration between instances of `WaxOn` and `WaxOff` is coordinated by invocations of `notifyAll` (lines 7 and 12) and `wait` (lines 18 and 24) in the class `Car`. These methods are used to suspend and restart threads while they are waiting for the condition of the waxing-polishing process to change. The boolean variables `waxOn` and `shiny`, declared in line 2 and 3, respectively, indicate the state of the waxing-polishing process. Since these state variables are shared across threads, access to them is coordinated using synchronization: note that all methods in `Car` are declared as `synchronized`. In `waitForWaxing` (lines 15 to 19), the variable `waxOn` is examined, and if it is `false`, the current thread is suspended by invoking `wait`; and the underlying lock is released. Releasing the lock is necessary because, to change the state of the waxing-polishing process, the lock must be acquired by each of the running threads in turn.

For example, when an instance of `WaxOn` calls `waxed` (lines 5 to 8), the lock must be acquired to change the variable `waxOn` to `true`. Then, `notifyAll` is invoked from within `waxed`, which wakes up the other thread waiting in the object's wait set. As a result, the other thread is eligible to acquire the lock on `Car` by invoking `buffed` (lines 10 to 13); setting the state of the waxing-polishing process to `false` before calling `notifyAll`. During execution, the two-step process involving `WaxOn` and `WaxOff` continues to iterate until the variable `shiny` is set to `true`.

Listing 6.11: `Car` class.

```
1  public class Car {
2    private boolean waxOn = false;
3    private boolean shiny = false;
4
5    public synchronized void waxed() {
6      waxOn = true; //ready to buff
7      notifyAll();
8    }
9
10   public synchronized void buffed() {
11     waxOn = false; //ready for another coat of wax
12     notifyAll();
13   }
14
15   public synchronized void waitForWaxing()
16       throws InterruptedException {
17     while (waxOn == false)
18     wait();
19   }
20
21   public synchronized void waitForBuffing()
22         throws InterruptedException {
23     while (waxOn == true)
24     wait();
25   }
26
27   public synchronized boolean isShiny() {
28      return shiny;
29   }
30
31   public synchronized void setShiny(boolean b) {
32     shiny = b;
33   }
34 }
```

To execute this example, the class shown in Listing 6.12 was implemented. `WaxOMatic` performs the necessary actions to initialize and start the elements involved in this example. First, an instance of `Car` is created in line 5. Second, a `Thread` object is initialized, passing an instance of `WaxOff` as its target object (line 6). Then, the thread's name is set to

“Buffer-1” in line 7. Third, from lines 8 to 9 all the same setup steps are performed, but this time for an instance of `WaxOn` and its respective `Thread` object. In line 9, the name of the `Thread` object responsible for invoking `WaxOn`’s `run` method is set to *“Waxer-1”*.

After the initialization step, the previously instantiated thread objects are started in lines 10 and 11. Next, the instruction in line 12 causes the current thread (main) to suspend execution for ten seconds, giving the other two threads (i.e., `bufferThread` and `waxerThread`) a chance to perform their tasks. When the main thread runs again, it concludes the waxing-polishing process by calling the `setShiny` method on `Car` (line 13).

Listing 6.12: Class containing the `main` method, which declares, initializes, and starts the threads involved in the example.

```
1  import java.util.concurrent.TimeUnit;
2
3  public class WaxOMatic {
4    public static void main(String[] args) throws Exception {
5      Car car = new Car();
6      Thread bufferThread = new Thread(new WaxOff(car));
7      bufferThread.setName(‘‘Buffer-1’’);
8      Thread waxerThread = new Thread(new WaxOn(car));
9      waxerThread.setName(‘‘Waxer-1’’);
10     waxerThread.start();
11     bufferThread.start();
12     TimeUnit.SECONDS.sleep(10); //run for a while...
13     car.setShiny(true);
14   }
15 }
```

Running the previous code prints out something similar to the output shown in Figure 6.2. The command used to run the example program is shown in the first line of Figure 6.2, ❶. Such a command line can be broken down in four parts:

(i) `max` is used to launch Maxine VM from the command line[5];

(ii) `-DrecordTo` specifies the path (directory) and the name of the file to which the synchronization sequence of the current execution is to be saved. This Java property is used in record, replay, and exploration mode;

(iii) `-DplayFrom` specifies the path (directory) and the name of the file containing the synchronization sequence to be enforced by the HLL VM. This Java property is used only in replay and exploration mode.

[5]As of the time of this writing, in the Maxine VM project, the `max` script was ported to Python in order to improve portability. The Python script is named `mx.py`, so now `mx` is used to launch Maxine VM instead.

(iv) `WaxOMatic` represents the name of the class to execute.

When the program is run, our implementation creates the file specified by the `recordTo` property. If the file in question already exists, it is overwritten. As for the second property, since the file `play.txt` does not exist, Maxine VM runs in record mode and no file is generated. During execution, control goes back and forth between the waxing and the polishing threads, which results in the output pointed out in Figure 6.2, ❷. After `shiny` is set to `true`, both threads finish execution, printing out the messages indicated by ❸ in Figure 6.2.

```
                          WaxOMatic Output
❶ $ max -DrecordTo=./syncsequence.txt -DplayFrom=./play.txt WaxOMatic
                                   ⎧ Wax on!
                                   ⎪ Wax off!
                                   ⎪ Wax on!
                                   ⎪ Wax off!
❷ repeats for about 10 seconds     ⎨ ...
                                   ⎪ Wax on!
                                   ⎪ Wax off!
                                   ⎪ Wax on!
                                   ⎩ Wax off!
                                   ⎧Ending Wax On task
❸ shows that the process is over   ⎩Ending Wax Off task
```

Figure 6.2: Output of `WaxOMatic` when it is run for the first time. That is, when there is no previously recorded synchronization sequence to enforce.

At run time, our implementation instruments the program before it is compiled by Maxine VM's JIT compiler. Code regions guarded by mutual-exclusion locks are modified as described in Section 6.3. In this example, the only class containing synchronized methods is `Car`.

6.4.1 The Synchronization Sequence Domain Specific Language

The main purpose of executing programs in record mode is to capture synchronization sequences. Our implementation records synchronization sequences in a domain specific language (DSL) (Sprinkle et al., 2009) designed to express these events in a succinct fashion.[6] The beginning of the synchronization sequence resulted from running the example

[6]Alternatively, we could have used Extensible Markup Language (XML) files to store information about synchronization sequences. However, we decided that using an XML parser to manipulate synchronization sequences would be an overkill.

program is shown in Listing 6.13.[7] Line 1 indicates that *"Buffer-1"* executes first and, upon invoking `isShiny`, it acquires the lock on the shared instance of `Car`.[8] The thread proceeds to call `waitForWaxing`, which checks if the `boolean` variable `waxOn` is `false`. At this point in the execution, `waxOn` is `false`, causing the current thread to invoke `wait`. As a result, it releases the lock on the shared instance of `Car` and enters the wait set of this instance (Listing 6.13, line 2). As shown in Listing 6.13, threads in the wait set of a certain object or class appear after `WS`.

Listing 6.13: Part of the synchronization sequence recorded during the execution of the example program.

```
1  Buffer-1 locked on Car<instance>(WS:);
2  Buffer-1 entered wait set of Car<instance>(WS: Buffer-1);
3  Waxer-1 locked on Car<instance>(WS: Buffer-1);
4  Waxer-1 notified all threads waiting on Car<instance>(WS: Buffer-1);
5  Buffer-1 left wait set of Car<instance>(WS:);
6  Waxer-1 released lock on Car<instance>(WS:);
7  Waxer-1 locked on Car<instance>(WS:);
8  Waxer-1 entered wait set of Car<instance>(WS: Waxer-1);
9  Buffer-1 locked on Car<instance>(WS: Waxer-1);
10 Buffer-1 released lock on Car<instance>(WS: Waxer-1);
11 Buffer-1 locked on Car<instance>(WS: Waxer-1);
12 Buffer-1 notified all threads waiting on Car<instance>(WS: Waxer-1);
13 Waxer-1 left wait set of Car<instance>(WS:);
14 Buffer-1 released lock on Car<instance>(WS:);
15 Buffer-1 locked on Car<instance>(WS:);
16 Buffer-1 entered wait set of Car<instance>(WS: Buffer-1);
17 ...
```

"Waxer-1" executes next, acquiring the lock on the `Car` instance upon invoking `isShiny` (Listing 6.13, line 3). Then, it prints "Wax on!" to the output. The next instruction causes the current thread to suspend execution for 200 milliseconds. However, since no other thread is eligible to execute (the main thread is also suspended and *"Buffer-1"* is in the wait set of the `Car` object) and *"Waxer-1"* still has ownership of the lock, nothing happens until *"Waxer-1"* resumes execution. *"Waxer-1"* continues execution by invoking `waxed`, which sets `waxOn` to `true` and calls `notifyAll`. Invoking `notifyAll` removes *"Buffer-1"* from the wait set, thereby making it eligible to be executed again (Listing 6.13, lines 4, 5,

[7]Note that our DSL uses fully qualified names for classes. However, for brevity sake, in this example we omitted package names.

[8]Actually, for the sake of brevity, synchronization events related to the invocation of `isShiny` are omitted since they do not contribute much to the discussion. Each invocation simply adds one lock-acquisition event and one lock-release event to the synchronization sequence. Therefore, we are considering that the lock is acquired when the current thread invokes `isShiny` and that the running thread holds the lock until returning from another `synchronized` method invocation; be it any method but `isShiny`.

and 6). Next, *"Waxer-1"* calls `waitforBuffing`. Inside `waitForBuffing`, *"Waxer-1"* calls `wait`, releasing the lock on the `Car` instance, suspending itself, and entering the underlying wait set (Listing 6.13, lines 7 and 8).

When *"Buffer-1"* tries to resume execution from where it left off, it has to re-acquire the lock. Only after re-acquiring the lock it can return from `waitForWaxing`, as shown in Listing 6.13, line 9. The lock is released as soon as *"Buffer-1"* returns from `waitForWaxing`, as indicated in Listing 6.13, line 10. Afterwards, *"Buffer-1"* prints "Wax off!" to the output and sleeps for 200 milliseconds; meanwhile, *"Waxer-1"* is still in the wait set. Once this time is elapsed, *"Buffer-1"* has to acquire the lock again to invoke `buffed` (Listing 6.13, line 11). Within the method body, `waxOn` is set to `false` and `notifyAll` is invoked. The call to `notifyAll` removes *"Waxer-1"* from the wait set (Listing 6.13, lines 12 and 13). In addition, since `waxOn` was set to `false`, *"Buffer-1"* is unable to continue its execution after invoking `waitForWaxing`, as indicated in line 16. Synchronization events similar to the aforementioned ones continue to be captured until the main thread wakes up and sets `shiny` to `true`. The size of the resulting file is less than 38KB and it contains 400 lines.

6.4.2 Replaying a Synchronization Sequence

As mentioned earlier, our implementation provides support for replaying the execution of multithreaded programs. This is achieved in replay mode, which forces the monitored program to execute according to a given synchronization sequence. Synchronization sequences are enforced by the instrumentation discussed in Section 6.3. When in replay mode, our implementation reads the synchronization sequence from a file created at the end of the record mode.

Specifically, we consider that a synchronization sequence differs from another synchronization sequence when the order in which they access synchronized code blocks is different. Controlling all accesses to synchronized blocks results in poor concurrent performance: when our implementation enforces synchronization sequences and multiple threads contend for the same lock, throughput suffers.

In the light of this instrumentation overhead, in replay mode programs usually take longer to run, which may further complicate the replay of time-bound executions. Therefore, in order to properly replay executions, it is necessary to either take a conservative approach and increase their execution spans and disable any time limitations (see Section 6.6). For instance, line 12 of Listing 6.12 shows that the main thread sleeps for about ten seconds, which gives the other threads roughly the same time to execute. After that, the main thread sets the variable `shiny` to `true`, which causes the other two threads to terminate their executions (see the `while` loops in Listings 6.9 and 6.10). One way of replaying the example program is to set the running time to around 45 seconds.

When our implementation reaches the end of the given synchronization sequence, the threads stall. Thus, considering the execution of the example program in replay mode, two scenarios are possible:

- `shiny` is set to `true` before the whole synchronization sequence is replayed. That is, the execution time indicated in line 12 of Listing 6.12 expires before the reproduction of the whole synchronization sequence. In this case, only a subset of the synchronization sequence is replayed.
- the reproduction of the synchronization sequence takes place before the execution time expires. This is the optimal case scenario for replaying time-bound programs.

6.4.3 Exploring Interleavings

As can be seen in Listing 6.13, our DSL does not represent all possible schedules. Instead, it focuses on logging the interleavings that take place when threads contend for synchronized blocks. As mentioned in Subsection 6.3.1, our implementation allows the exploration of these synchronization sequences captured during record mode in hopes of finding error-yielding interleavings.

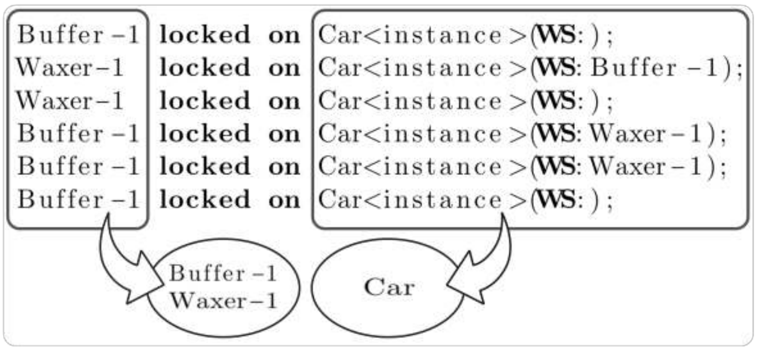
Algorithm 6.1 gives an overview of the main steps by which our implementation uses a previously recorded synchronization sequence to expose new possible interleavings. As shown in line 2, only the lock-acquisition events are taken into account. After filtering out the other synchronization events, the algorithm analyzes the right hand side of the lock-acquisition events in order to identify the objects used as locks (line 3). Similarly, the left hand side of the synchronization events is investigated to identify what threads tried to acquire a certain lock (line 5).

Naturally, there can be an arbitrary number of threads and shared locks. Our algorithm groups threads into sets according to the lock that they acquire at run time (line 6). Given a synchronization sequence file containing a lock L_1, which is shared by three threads, T_1, T_2, and T_3, our algorithms yields two sets: one comprised only of the lock, $L = \{L_1\}$, and the other containing the threads, $T = \{T_1, T_2, T_3\}$. This step of the algorithm is further exemplified in Figure 6.3.

Figure 6.3 shows the synchronization file in Listing 6.13 after being processed to remove all synchronization events but lock-acquisition ones, as illustrated in line 2 of Algorithm 6.1. After performing the step described in line 3, our algorithm identifies that the only lock being used throughout the file is the one in `Car`. Then, when screening each synchronization event for threads that contend for the lock (Algorithm 6.1, line 5), two threads are identified: *“Buffer-1”* and *“Waxer-1”*.

Algorithm 6.1 Interleaving exploration.

```
 1: procedure INTERLEAVINGEXPLORATION($synchSequence$)
 2:     $lockAcqEvents \leftarrow synchSequence.filterBy(ACQUIRE\_LOCK)$
 3:     $locks \leftarrow dslParser.parseRightHandSide(lockAcqEvents)$
 4:     for each lock in locks do
 5:         $threads \leftarrow dslParser.getThreadsThatContendFor(lock, lockAcqEvents)$
 6:         $threadsAndLockHash \leftarrow \{lock => threads\}$
 7:     end for
 8:     for each key, values in threadsAndLockHash do
 9:         for each thread in values do
10:             $interleavings \leftarrow [thread, key]$
11:         end for
12:     end for
13: end procedure
```



```
Buffer-1  locked on Car<instance>(WS: );
Waxer-1   locked on Car<instance>(WS: Buffer-1);
Waxer-1   locked on Car<instance>(WS: );
Buffer-1  locked on Car<instance>(WS: Waxer-1);
Buffer-1  locked on Car<instance>(WS: Waxer-1);
Buffer-1  locked on Car<instance>(WS: );
```

Figure 6.3: Grouping threads according to the lock that they contend for.

Locks and their respective contending threads are stored in a hash map whose keys are the lock themselves and the values are arrays containing the threads as indicated in line 6. Subsequently, the interleavings that we wish to explore are represented as a Cartesian product of these two sets: $T \times L = \{T_1, T_2, T_3\} \times \{L_1\} = \{(T_1, L_1), (T_2, L_1), (T_3, L_1)\}$.

The exploration of interleavings starts from the last lock-acquisition and proceeds backward to the first one. In exploration mode, the execution is replayed until the lock-acquisition event that is predetermined to change. Each execution explores only one new interleaving. For instance, considering Figure 6.3, during the first execution in exploration mode, our implementation will try to enforce a scenario in which *“Waxer-1”* acquires the lock on `Car` instead of *“Buffer-1”*. In other words, each permutation resulted from {*“Buffer-1”*, *“Waxer-1”*} × {`Car`} is enforced in turn.

Most of the steps described in Algorithm 6.1 are carried out or triggered by a class named `ExplorationController`. When a program is run for the first time in exploration mode, `ExplorationController` triggers the parser to create the hash map containing the locks and their respective contending threads. The interleavings that will be explored during execution are derived from such a hash map. Apart from storing the interleavings that should be enforced, `ExplorationController` keeps track of the interleaving exploration by means of an internal counter whose value indicates the current line in the synchronization sequence that is being explored. Thus, the initial value of this counter corresponds to the number of lines of the synchronization sequence file being explored. Upon exploring all possible combinations for a given line, such a counter is decreased by one. Object serialization (Kurotsuchi, 1997) is used to persist `ExplorationController`'s data between executions.

Note that it is not possible to enforce all interleavings because Algorithm 6.1 at times generates infeasible interleavings. When our implementation fails to enforce an interleaving, it signals that the exploration of that particular interleaving has been tampered with, and it resumes execution in record mode.

6.5 The Interplay Between Synchronization Sequence Files and Each Execution Mode

Figure 6.4 shows an overview of the role that synchronization sequence files play in each execution mode. Essentially, two configuration properties determine which execution mode will take place, as previously indicated in Figure 6.2. If no `playFrom` property is supplied,[9] our implementation runs in record mode. Running a program in record mode logs the synchronization sequence that took place during execution in the file specified by the `recordTo` property, as shown in Figure 6.4.

In order to execute a program in replay mode, it is mandatory to specify a valid synchronization sequence file through the `playFrom` property. As illustrated in Figure 6.4, replaying a synchronization sequence does not alter the contents of the file. Optionally, a copy of the synchronization sequence file being replayed can be created: when `playFrom` and `recordTo` specify different files, the contents of the file indicated by the former are copied to the file indicated by the latter. In Figure 6.4, for example, `A` and `B` are two different files with the same synchronization sequence.

To run a multithreaded program in exploration mode, the value of the `playFrom` property has to point to a valid synchronization sequence file. To distinguish this execution mode from replay, an extra configuration property is employed. Such a property, called `interExplor`, has to be set to `true` (as shown in Figure 6.5). In exploration mode, the

[9]The same happens when the file indicated by the `playFrom` property does not exist or is empty.

synchronization sequence file indicated by `playFrom` is enforced until execution reaches the line whose number corresponds to the current value of `ExplorationManager`'s counter. Upon executing this line, `ExplorationManager` tries to execute a different synchronization event. Next, execution resumes in replay mode. As shown in Figure 6.4, in exploration mode, it is also possible to save the new synchronization sequence to a file simply by specifying an empty file through the `recordTo` property. At its simplest, when the exploration of a new interleaving succeeds, the copy ($\mathtt{A}_1 \ldots \mathtt{A}_n$) differs from the original synchronization sequence (`A`) in exactly one line (Figure 6.4). As depicted in the bottom part of Figure 6.4, the interleaving exploration continues until all lines of the original file have been explored, each execution explores a new interleaving.

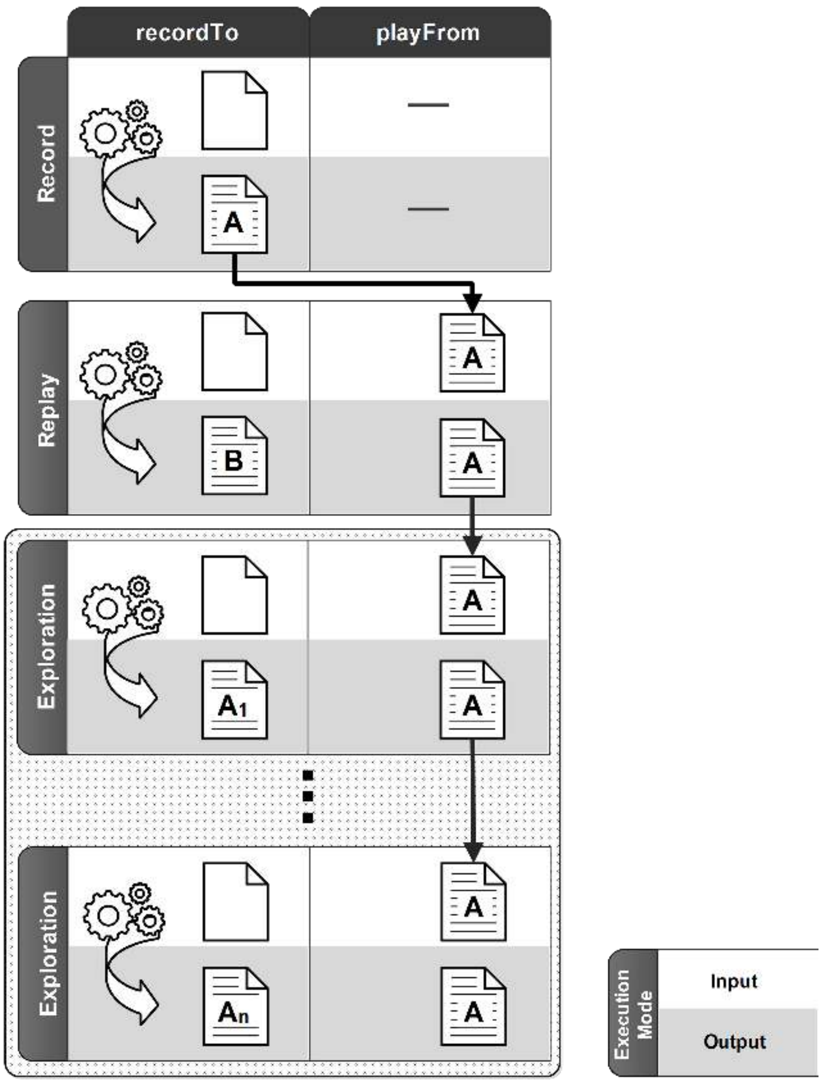


Figure 6.4: The interplay between execution modes and synchronization sequence files. A valid synchronization sequence is used as input to all execution modes but record. In record mode the `recordTo` property indicates the file that will contain the main output, whereas in other execution modes it is used to log debugging information.

To explore new interleavings manually, testers have to edit synchronization sequence files and then use the replay mode in hopes of enforcing the new scheduling encoded into these files.

```
Exploration Mode
$ max -DrecordTo=./A1.txt -DplayFrom=./A.txt -DinterExplor=true ProgName
```

Figure 6.5: Triggering the exploration mode from the command line.

6.6 Evaluation

In this section, we evaluate our proof-of-concept implementation on a set of multithreaded Java programs. The evaluation is twofold. Firstly, we examine the record and replay mode of our implementation by gauging the overhead incurred by their respective instrumentations. Secondly, we shed some light on how the resulting DSL files can be modified by testers of multithreaded Java programs to expose resource deadlocks (Goetz et al., 2006).

This evaluation was carried out using Maxine VM on a 2.1GHz Intel Core 2 Duo with 4GB of physical memory running Mac OS X 10.6.6. We used ten Java programs ranging in size from 44 to 143 lines of code, totaling 885 lines of code. The two largest programs are comprised of five Java files. During the selection of these programs, we gave priority to textbook examples. As listed in Table 6.1, all subject programs are taken from Java textbooks. `Producer-Consumer` (Niemeyer and Knudsen, 2005), for instance, implements the classic producer-consumer problem, which has been extensively used to exemplify how to synchronize multiple threads that share a common resource such as a fixed-size buffer. Programs that were originally designed to run in an infinite loop were modified to run for a fixed period of time.

Aside from the instrumentation overhead in question, the performance of Java programs is influenced by several factors. For example, GC and JIT compilation are two factors that impact performance. For the sake of mitigating these sources of variability and thus carrying out a more rigorous performance evaluation, the measured running times reported in Table 6.2 are the average (mean) execution time across 3 runs; with the exception of time-bound programs, which had their maximum execution time set to ten seconds both during execution without instrumentation and in record mode. For the subject programs whose number of synchronization events varies during execution, resulting in varying-sized synchronization sequences, we report the median size of the synchronization sequence file and its respective number of lines across 3 runs.

As shown in Table 6.2, the overhead imposed by the record mode instrumentation is negligible. The highest overhead measured was 25.81% (`SynchronizedInteger`). As for the other subject programs, the overheads in record mode were 0.98% (`Counter`), 3.58% (`AccountDanger`), and 5.82% (`Bank & Company`). Interestingly, the instrumented version of `SynchronizedBuffer` performed better in record mode than its non-instrumented version. It took on average 18.93s to execute in record mode and 19.77s to run without instrumentation. This suggests that when running in record mode our approach is minimally intrusive. We conjecture that this is due to the fact that, when replay is not required, guard conditions that prevent threads from acquiring locks are relaxed. This also indicates that the record instrumentation can be kept switched on at all times. Doing so can contribute to pinpoint and eliminate Heisenbugs: whenever an error occurs, it is possible to gather further information about its cause by analyzing the generated synchronization sequence file.

Table 6.1: Multithreaded programs extracted from the literature.

Program	**Lines of Code**[†]	**Files**	**Textbook**
`SynchronizedInteger`	44	2	(Goetz et al., 2006)
`AccountDanger`	62	2	(Sierra and Bates, 2008)
`Counter`	76	2	(Goetz et al., 2006)
`StackImpl`	76	2	(Mughal and Rasmussen, 2008)
`Bank & Company`	77	4	(Javier, 2012)
`Producer-Consumer`	92	3	(Niemeyer and Knudsen, 2005)
`Dining Philosophers`	92	4	(Garg, 2004)
`WaxOn WaxOff`	94	4	(Eckel, 2006)
`Reader-writer`	129	5	(Silberschatz et al., 2009)
`SynchronizedBuffer`	143	5	(Deitel and Deitel, 2004)
Total	885	33	

[†]Physical lines of code (non-comment and non-blank lines).

Due to the different characteristics of the selected programs, the amount of synchronization events yielded by them differs. For instance, `AccountDanger` and `Bank & Company` take roughly the same time to execute, however, the number of synchronization events generated by the latter is 20 times larger than the number of synchronization events produced by the former (Table 6.2). Both `AccountDanger` and `Bank & Company` generate only lock-acquisition and lock-release events. In other words, during the execution of these programs there are no calls to `wait`, so none of the running threads enters a wait set. In comparison, `SynchronizedBuffer` generates all types of synchronization events that can be captured by our DSL at run time, and yet the resulting synchronization sequence file is smaller than the one generated for `Bank & Company`.

In contrast to running applications in record mode, enforcing synchronization sequences in replay mode results in noticeable overhead. In order to properly replay the execution of time-bound applications, two approaches can be taken: *(i)* increasing the running time limit so that it takes into consideration the slowdown introduced by the replay instrumentation or *(ii)* turning off the time limit making our implementation halt execution after reproducing the last synchronization event. During this evaluation, we took the second approach. The mean replay times in Table 6.2 suggest that the replay instrumentation incurs moderate overhead. We conjecture that some of this slowdown can be ascribed to the additional interthread coordination cost introduced by our instrumentation. One factor that seems to influence the time our implementation takes to replay an execution is the number of involved threads. It seems that the more threads involved, the longer it takes to replay an execution.

Table 6.2: Performance of our implementation in record and replay mode.

Program	Threads	Exec. Time*	Record⋆	Replay⋆	File Size†	Lines‡
`SynchronizedInteger`	2	0.31s	0.39s	0.48s	0.82K	8
`AccountDanger`	2	2.51s	2.60s	3.31s	1.8K	20
`Counter`	2	10.15s	10.25s	23.09s	93K	1014
`StackImpl`	2	≈10s	≈10s	15.10s	1.7K	19
`Bank & Company`	2	2.06s	2.18s	4.01s	41K	400
`Producer-Consumer`	2	≈10s	≈10s	18.35s	4.4K	45
`WaxOn WaxOff`	2	≈10s	≈10s	39.53s	38K	400
`SynchronizedBuffer`	2	19.77s	18.93s	27.01s	9.5K	78
`Dining Philosophers`	5	≈10s	≈10s	55.21s	11K	80
`Reader-writer`	6	≈10s	≈10s	89.67s	87K	889

*Average execution time across 3 runs with instrumentation turned off.

⋆Average execution time across 3 runs in either record or replay mode.

†File size indicates the amount of storage space required to store the synchronization file.

‡Values in this column represent the number of lines in synchronization sequence files.

6.6.1 Editing Synchronization Sequence Files to Expose Interesting Interleavings

Given that our implementation relies on synchronized blocks and methods, it might not be very effective to detect race conditions. Simply put, programs that properly synchronize access to data are likely to be free of data races. Our implementation, however, can be useful to simulate deadlock scenarios in programs whose outcomes depend on the execution order. Aimed at demonstrating this, we manually modified the synchronization sequence file generated for `Dining Philosophers` to make it run into a deadlock.

As pointed out by Garg (2004), the dining philosophers problem was first formulated and solved by Dijkstra. This synchronization problem illustrates the need to properly allocate shared resources among threads in a deadlock-free and starvation-free fashion (Silberschatz et al., 2009). Deadlock arises when each philosopher (thread) grabs the chopstick (shared resource) to his left and then has to wait for the adjacent philosopher to release the other chopstick. This behavior can be reproduced by inserting the synchronization events shown in Listing 6.14 into a previously recorded synchronization sequence. The code in Listing 6.14 forces each philosopher to grab only the chopstick on his right, thus none of them can resume execution because they have to wait for the left chopstick, which is held by the philosopher to their left.

Listing 6.14: Snippet of code that causes `Dining Philosophers` to run into a deadlock. This code has to be inserted before any lock-acquisition related event.

```
...
 Phil-4 locked on BinarySemaphore<instance>(WS:);
 Phil-3 locked on BinarySemaphore<instance>(WS:);
 Phil-2 locked on BinarySemaphore<instance>(WS:);
 Phil-1 locked on BinarySemaphore<instance>(WS:);
 Phil-0 locked on BinarySemaphore<instance>(WS:);
...
```

Tools such as ConTest (Edelstein et al., 2003) increase the probability of the aforementioned concurrency hazard by trying to introduce a context switch after each left chopstick is picked. Using our implementation, the same can be achieved in a deterministic way by editing the DSL file generated from running `Dining Philosophers`. We were able to simulate the aforementioned scenario in which none of the involved threads is able to resume execution.

6.7 Known Limitations

Capturing synchronization sequences from non-terminating programs poses a challenge due to the infinite size of these synchronization sequences. A possible workaround is to capture only the first n synchronization events. Likewise, debugging long running programs using our implementation is ineffective due to huge synchronization sequences and the overhead associated with enforcing them.

In the current implementation, deadlocks might occur during replay mode or when exploring new interleavings if one of the involved threads terminates abnormally (e.g., due to an uncaught exception). In the future, we plan to address this by wrapping our instrumentation into a `try-catch` block,[10] which will enable our implementation to properly resume execution. In record mode, however, it is possible to capture synchronization sequences that lead to deadlock. In other words, despite the fact of having to force the JVM to shut down in case of deadlock, the synchronization sequence captured up to the point of such a concurrency hazard is not lost. A shutdown hook (Goetz et al., 2006) is used to ensure that synchronization sequences are saved to file when the JVM is shutting down.

Prior to Java 5, the only mechanisms for coordinating access to shared resources were synchronized methods and blocks and the `volatile` keyword (Goetz et al., 2006). Java 5 introduced a new option, namely, `ReentrantLock`.[11] Despite the fact that `ReentrantLock` should not be seen as a replacement for intrinsic locking, `ReentrantLock` implements reentrant mutual exclusion lock with the same basic behavior as synchronized methods and blocks, but including additional advanced features. As of this writing, our implementation does not support the replay of programs that use `ReentrantLock` in lieu of synchronized blocks or methods.

Similarly to enforcing executions in replay mode, exploring different synchronization sequences significantly slows down our implementation. Another source of slowdown is that the number of possible synchronization sequences increases exponentially with the number of threads and number of acquire-lock events.

Given that our implementation relies on bytecode instrumentation, it does not support native methods.

6.8 Related Work

As previously mentioned, testing multithreaded programs is as challenging as multithreaded programming. Part of this complexity can be ascribed to the shortcomings of unit testing frameworks, which have been developed with single-threaded programming in mind. Therefore, they fail to take into account issues that arise when testing concurrent programs. To cope with these issues, researchers have been extending test frameworks to include basic facilities that support the creation and execution of unit tests for programs containing multiple threads.

[10]Another possibility is to use the uncaught exception handlers provided by the `Thread` API through the `UncaughtExceptionHandler` interface. Handlers that implement this interface are invoked when any of the registered threads abruptly termine (`http://docs.oracle.com/javase/1.5.0/docs/api/java/lang/Thread.UncaughtExceptionHandler.html`).

[11]`http://docs.oracle.com/javase/1.5.0/docs/api/java/util/concurrent/locks/ReentrantLock.html`

An example of shortcoming is that test frameworks tailored to single-threaded programs ignore failures in auxiliary threads. ConcJUnit is a concurrency-aware version of JUnit that supports the developer in writing unit tests for concurrent programs (Ricken and Cartwright, 2009). By handling uncaught exceptions and failed assertions in all threads, ConcJUnit simplifies the creation of unit tests for concurrent programs. In addition, it detects child threads that were not forced to terminate before the main thread finishes execution.

Other frameworks have focused on recording failing interleavings and deterministically replaying them by transforming the program under test. Carver and Tai (1991) use a language-based transformation approach that turns the original program under test into an equivalent program that employs semaphores and monitors to control synchronization.

Writing large test cases containing many possible interleavings is an established strategy for testing concurrent programs. Thus, aimed at increasing the chances of discovering faulty interleavings, several concurrent test frameworks provide facilities to boost the range of possible interleavings. Edelstein et al. (2002) developed a tool for detecting synchronization faults named ConTest. To reveal concurrent faults, ConTest seeds the program under test with various primitive calls (e.g., sleep and yield) at shared memory accesses and synchronization events. Then, at run time, ConTest randomly decides whether the seeded primitives are to be executed, increasing the probability of uncovering faults. Along the same lines, CalFuzzer (Joshi et al., 2009) and CTrigger (Park et al., 2009) employ analysis techniques to generate potentially erroneous schedules.

Besides these frameworks, researchers have been developing record and replay mechanisms, as the ones discussed in the previous chapter. A notable example of such mechanism is DEJAVU (Choi et al., 2001), which is similar to our implementation because it was implemented as part of a JVM. However, given that DEJAVU was implemented in a green-thread JVM, it achieves a greater degree of control over threads. Instead of relying on a scheduler, our workaround solution capitalizes on the way threads are implemented in the JVM and its intermediate language. Differently from our implementation, DEJAVU deterministically replays the entire execution, not only the order in which synchronization events take place. When it replays a program up to a synchronization primitive, it also faithfully replays the whole program and execution environment states. Compared to DEJAVU, our implementation records much less run-time information. Most record and replay mechanisms sit on top of execution environments. An example of record and playback tool is RecPlay (Ronsse and De Bosschere, 1999).

6.9 Concluding Remarks

Designing and implementing concurrent programs is difficult. Most of this complexity stems from the inherent non-determinism of concurrent programs and the shortcomings of the abstractions used for representing concurrency (Lee, 2006). To overcome the nondeterminism of multithread Java programs, our implementation piggybacks on the way that the execution environment serializes access to certain blocks of code.

Given the granularity with which our proof-of-concept implementation deals with non-determinism, executions are replayed in terms of coarse-grained synchronization events, i.e., accesses to synchronized blocks. Thus, synchronization events that take place outside synchronized regions are not taken into account.

We believe that recording synchronization sequences into a reader-friendly textual format and deterministic replay are useful to understanding and debugging concurrency-related faults.

Our approach and implementation thereof still have room for improvements. In the future, to mitigate the slowdown caused by the introduced instrumentations, we intend to investigate the idea of running multiple Maxine VM instances in a concurrent fashion, each exploring different synchronization sequences.

Chapter

7

Conclusions and Future Work

This chapter revisits the research problem and the RQs (Section 7.1). Drawing on the findings of our investigation, we further elaborate on the advantages and disadvantages of extending JIT compilers to support software testing techniques (Section 7.2). This chapter also restates the contributions (Section 7.3), discusses the current limitations (Section 7.4) of our proof-of-concept implementation, and suggests future research directions (Section 7.5). The chapter concludes with a discussion of the relevance of this research and a summary of the technical challenges faced when extending Maxine VM (Section 7.6).

7.1 Research Problem and RQs Revisited

Our survey of the literature, along with our own experiences, led us to believe that the bulk of the research on HLL VMs has focused on developing faster JIT compilers and GCs. Few efforts have tried to augment HLL VMs with software testing support.

We argued that lowering the implementation of testing support into the HLL VM could offer several advantages. First, it would open up possibilities for further speeding up computationally expensive software testing techniques. In other words, software testing support would better leverage off all the optimizations present in today's HLL VMs when integrated within HLL VMs. Second, it would be possible to achieve extensive control over program execution by taking advantage of the facilities afforded by HLL VMs. So, this dissertation tackled the research problem of investigating whether HLL VMs provide a sound basis for implementing software testing support. Based on this research problem, two RQs were formulated. In the light of our findings, these RQs are answered as follows.

- $\mathbf{RQ_1}$**:** Can HLL VMs be harnessed to support software testing?
  - We addressed $\mathbf{RQ_1}$ by designing and implementing a testing infrastructure within a modern HLL VM. We started off with the selection of the target HLL VM. Given that the results of our systematic mapping suggest that JVMs are the most widely used HLL VMs, we settled on using Maxine VM, which is a mature, research-oriented, high-performance JVM implementation. Next, we set out to demonstrate the feasibility of capitalizing on HLL VMs to speed up compute-intensive testing techniques. To this end, we chose to retrofit mutation testing support to Maxine VM. The resulting implementation that automates weak mutation was described in Chapter 4. Subsequently, facilities for supporting the test of multithreaded Java programs were also integrated within the chosen JVM. Such facilities include record-and-playback and an interleaving exploration mechanism, which were presented in Chapter 6. Therefore, the answer to $\mathbf{RQ_1}$ is that HLL VMs can be harnessed to support software testing.

As we looked at the results of retrofitting software testing support into Maxine VM, we fine-tuned our investigation by asking:

- $\mathbf{RQ_2}$: What sort of software testing support is more suited to modern HLL VMs?
  - Our results suggest that the benefits provided by modern HLL VMs are suited to speed up the execution of computationally expensive techniques such as mutation testing. On the other hand, controlling threads from within Maxine VM has proved challenging. Essentially, the problem lays within the thread model used by most high-performance HLL VMs. Maxine VM and other contemporary JVMs are inappropriate for hosting record-and-playback mechanisms because they use native threads. That is, these HLL VMs rely on the OS to manage threads, which are compiled to pthreads. HLL VMs tailored to resource-constrained devices use green threads. In a green thread model, the implementation of threads is managed entirely in the HLL VM. Therefore, we surmise that green-thread HLL VMs might afford better control over threads. Being able to access and modify the thread scheduler would make the implementation of facilities as the one described in Chapter 6 easier.

    Summing up, our findings indicate that the answer to $\mathbf{RQ_2}$ is that testing techniques that benefit from improved performance are better suited to be implemented as part of a cutting-edge execution environment. From within such an execution environment it is easier to capitalize on the optimization infrastructure (e.g., JIT compilation and GC). Additionally, by integrating those software testing features into the runtime environment, it is possible to take advantage of ahead-of-time compilation. That is, the set of classes that belongs to the core execution environment is usually compiled in an ahead-of-time fashion to generate a bootstrap image. Thus, adding testing features to the core of HLL VMs turns them into self-contained execution environments that require no dynamic class loading until later stages of execution. As for controlling threads, green-thread HLL VMs are more amenable to the implementation of replay mechanisms that entail a great degree of control over threads. Unfortunately, due to performance reasons, no mainstream JVM implementation uses green threading.

7.2 JIT Compilation: What Are the Advantages It Has to Offer to Software Testing Automation?

Based on the results of our investigation, we could draw the following conclusions. JIT compilers significantly increase the complexity of HLL VMs. Usually, HLL VMs need to be deployable on different platforms, so these runtime compilers have to be adapted for different hardware architectures. This makes JIT compilers difficult to implement and maintain, but the difficulty pales in comparison to the performance benefits they provide. Such a characteristic, along with the fact that it is easier to trigger the runtime compiler from within the execution environment, makes HLL VMs a sound basis for speeding up computationally expensive testing techniques. It turns out that taking advantage of Maxine VM's JIT compiler is fairly straightforward. However, modifying it seems to be quite complex, time-consuming, and error-prone.

As pointed out by Weyuker (2002), the analysis required by some code-based testing techniques as the dataflow criteria is similar to the one carried out by a JIT compiler: when looking for optimization opportunities, the compiler has to determine whether a given variable occurrence is either a definition or a use. Despite that, the complexity and brittleness in Maxine VM's compiler led us to believe that it might be hard to tailor it to even similar needs.

To circumvent the aforementioned problems, researchers that have extended JIT-enabled JVMs for some sort of code coverage analysis have opted for relying on bytecode transformation instead of modifying the runtime compiler (Chilakamarri and Elbaum, 2004). Apart from being a simple instruction set, bytecodes can be easily mapped to source code and some HLL VMs provide bytecode manipulation facilities. Therefore, we believe that it is more convenient to implement support for control flow and data flow criteria through bytecode instrumentation at loading time. In addition, if performance is not a concern, interpretive HLL VMs can be a better fit for tracking code coverage information and achieving a higher degree of control over execution. The main advantage of an interpretive HLL VM is that interpreters tend to be simpler than runtime compilers (Klint, 1981).

7.3 Summary of Contributions

We systematically surveyed the literature describing research into HLL VM. We found that most studies concentrate on improvements for boosting performance and introducing better GC capabilities. Furthermore, the results of this systematic mapping suggest that JVMs are by far the most investigated implementations within academic settings. These findings add to a growing body of literature on understanding trends in HLL VM research.

Our VM-integrated approach to weak mutation forks new threads to execute mutant methods. One of the benefits of forking new threads is that the program up until a method call does not need to be repeated for every mutant. This implements the split-stream execution envisioned by King and Offutt (1991). Moreover, the method model of Maxine VM was extended with mutation testing semantics, where certain methods are seen as mutants and can be in either of two states: alive or dead. The experiment shows that major savings in computational cost can be achieved by this weak mutation approach, speedups of up to 95% were obtained in comparison to muJava.

We retrofitted record-and-playback and interleaving exploration capabilities into Maxine VM. The contributions of this implementation are fourfold. First, our implementation is able to capture information about the order in which threads execute synchronized code and when they enter and exit the waiting set of shared objects. Second, using such an information, our implementation is able to ensure the deterministic re-execution of multithreaded Java programs. Third, our implementation can enforce the execution of new interleavings by modifying a given synchronization sequence file. Moreover, since synchronization sequences are expressed in a compact and user-friendly textual representation, testers can drive the exploration of new interleavings by changing these files by hand. Forth, we think that logging synchronization sequences into a reader-friendly representation is helpful for debugging concurrency-related faults. Our implementation is tailored to support approaches similar to the one proposed by Offutt et al. (1996b) (discussed in Section 5.4) whose feasibility and correctness are highly dependent on the degree to which deterministic re-executions and new scheduling sequences can be enforced.

Throughout this dissertation, the implementation details of the improvements to support software testing were discussed in terms of a JVM implementation. We argue that this is no limitation. We believe that most ideas discussed herein can be transferred into other execution environments. In fact, execution environments for other HLLs and their intermediate languages employ very similar concepts (Singer, 2003), thereby they can be extended in similar ways.

7.4 Limitations

While our proof-of-concept implementation bolsters our hypothesis that HLL VM can be augmented with testing support, its design has some limitations. This section briefly restates the main limitations, which have been discussed in detail in Sections 4.3 and 6.7.

- Limitations of the weak mutation system:
  - Our implementation does not automate the generation of mutant methods.
  - The execution of mutant native methods is not supported because these methods do not use Java stacks. Instead, native methods use C stacks (Liang, 1999). As a result, our implementation is not able to obtain a snapshot of the context before and after the invocation of native methods.
- Limitations of the record-and-playback mechanism:
  - Our implementation does not support the replay of programs that use `ReentrantLock` in lieu of synchronized blocks or methods.
  - Given that part of our implementation relies on bytecode transformations, it does not support native methods. Thus, programs with concurrent native code may not be properly replayed.
- Limitations of our interleaving exploration mechanism:
  - Exploring interleavings of long running programs using our implementation is ineffective. Although our approach minimizes the amount of data recorded (i.e., only coarse-grained synchronization events are logged), long running programs with lots of lock contention still may yield very large synchronization sequences. Given that no additional technique is used to reduce the size of the resulting synchronization sequence files, exploring interleavings as described in Subsection 6.4.3 may take too long: long running programs aggravate the state explosion problem.
- Limitations related to the integration of the split-stream execution and our replay mechanism:

  - In our split-stream approach, after the execution of an original method, new threads are forked to run mutants. When that happens, the main thread (i.e., thread executing the original method) is held until all mutants are run. In effect, the main thread is held before it returns from the original method (i.e., the frame of the original method is still the current frame when the thread is held). Since locks are released only after method completion, the threads spawned to run the mutants of a synchronized method are not able to acquire the lock associated with the underlying object (for instance methods) or class (for static methods), and thus they cannot execute the synchronized regions within these mutant methods. As a consequence, it is not possible to deterministically replay the execution of programs containing synchronized mutant methods. Two simple workarounds for this problem are discussed in Section 7.5.

- Limitations inherent to Maxine VM:
  - Since our implementation is part of Maxine VM, its architecture is less portable than similar tools. To be precise, while similar tools can run atop many JVM implementations, some elements of our implementation are specific to Maxine VM. Furthermore, our implementation can only run on the platforms that Maxine VM has been ported to. Currently, Maxine VM only supports 64-bit computer architectures; it runs on Linux, Solaris, and Mac OS.
  - Our implementation is not fully compatible with Java 7 programs. It is likely that this limitation will be removed in the future as the developers aim to keep Maxine VM up to date with respect to the latest version of the JVM specification (Wimmer et al., 2013). As of this writing, Maxine VM integrates with the JDK 7 class library. However, it does not implement the `invokedynamic` bytecode, introduced with Java 7, so Maxine VM cannot execute programs needing some Java 7 features.

7.5 Future Research Directions

We believe that the resulting HLL VM can be seen as an integrated testing environment that can be further tuned and augmented. Although fully functional, this integrated testing environment is subject to further improvements. The research described in this document can be continued in several directions:

- We intend to evaluate the scalability of our proof-of-concept implementations by carrying out follow-up experiments involving more complex programs. In these follow-up experiments we need to look at the following:

  - Regarding the weak mutation system, we need to examine how much of the achieved speed-up is due to multithreading (i.e., split-stream execution) and how much can be attributed to weak mutation. While we did not seek to answer this research question, it is relevant for the future.
  - In the experiment described in Section 4.2, we did not compare our variant of weak mutation with strong mutation in terms of effectiveness. It is known that weak mutation is not as effective as strong mutation. Future work can examine how the proposed method-based weak mutation compares with strong mutation in terms of effectiveness. Along similar lines, future work can also investigate more appropriate ways to implement weak mutation within Maxine VM. For example, instead of performing the state comparison after method executions, a finer-grained variant of weak mutation could compare the program states as follows: *(i)* after the first execution of the basic block that contains the mutated statement; or *(ii)* after the first execution of the mutated statement.

- There are three workarounds to make split-stream execution work in replay mode. The first workaround is to simply prevent synchronized methods from being mutated, which does not entail any changes to our implementation. The second workaround is to transform synchronized mutants into non-synchronized methods at loading time. However, it is worth noting that not every method can be turned into a regular method. For example, methods that invoke thread collaboration methods such as `notify` and `notifyAll` will not work properly when transformed into regular methods. The reason is that `notify` and `notifyAll` should only be called when the current thread owns the underlying object's lock, otherwise an `IllegalMonitorStateException` is thrown. The third workaround is modify our split-stream implementation so that the main thread releases the lock before forking new threads to execute mutants. Nevertheless, since only one thread (the one that owns the lock) would be able to be executed at any given time, this workaround defeats the purpose of the split-stream approach. Future work will be needed to investigate better ways to integrate split-stream and deterministic re-execution.
- The interleaving exploration algorithm can be improved with better heuristics.
- For a faster exploration of interleavings, our implementation can be extended so that it is possible to run multiple instances of Maxine VM in a concurrent fashion, each exploring a new interleaving.
- This research has shown that it is feasible and, in some cases, beneficial to implement testing techniques in an HLL VM. Aimed at creating an execution environment

hosting a wider variety of testing techniques, long term future work includes the investigation of how Maxine VM can be modified to support other testing techniques.

- More broadly, not only software testing techniques are susceptible to being integrated within HLL VMs. Future research should concentrate on examining programming mechanisms and constructs that are currently realized through code instrumentation or implemented in libraries but should rather be supported at HLL VM level.

7.6 Overall Conclusions

This research has supplied insights into the sort of software testing support that can capitalize on the features of state-of-the-art HLL VMs. This dissertation is the first to look at how HLL VMs can be extended with software testing support. Although previous research has built on compilers and interpreters to speed up mutation testing, ours is the first to explore a full-fledged, JIT-based HLL VM. The results of the experiment described in Section 4.2 further strengthened our hypothesis that the optimization infrastructure of HLL VMs has the potential of yielding marked speedup. Therefore, we posit that our findings advance knowledge and understanding in the field of mutation testing.

The implementation of the record-and-playback and interleaving exploration facilities was hindered by the fact that Maxine VM does not have a scheduler, which is a trait of modern JVMs. We argue that a greater degree of control over threads could have been achieved if we had modified a green-thread JVM. However, only JVMs tailored to small devices use green threading. Thus, choosing a JVM designed for small devices would limit the scope of our findings.

Despite Maxine VM's research-oriented design, modifying it posed some technical challenges. All in all, HLL VMs are complex and consist of many subsystems whose intricate interactions entail implicit dependencies; Maxine VM is no exception. Maxine VM has around 550,000 lines of code and includes more than ten subsystems. Thus, reasoning about which elements of these subsystems should be extended and the implications of modifications proved to be a challenge during the early stages of development. Another factor that significantly hindered the implementation of the two testing facilities was the slow turnaround time: getting feedback on whether changes to Maxine VM worked or not requires at least a five-minute wait for recompilation. This rendered the use of agile practices such as test-driven development (TDD) impractical.

Bibliography

ACREE, A. T. *On Mutation.* Ph.D. Thesis, Georgia Institute of Technology, 1980.

ADL-TABATABAI, A.-R.; KOZYRAKIS, C.; SAHA, B. Unlocking Concurrency. *Queue*, volume 4, number 10, pages 24–33, 2006.

ADVE, S. V.; HILL, M. D.; MILLER, B. P.; NETZER, R. H. B. Detecting Data Races on Weak Memory Systems. *ACM The ACM Special Interest Group on Computer Architecture (SIGARCH) Computer Architecture News*, volume 19, number 3, pages 234–243, 1991.

AGOSTA, G.; REGHIZZI, S. C.; SVELTO, G. Jelatine: A Virtual Machine for Small Embedded Systems. In: *Proceedings of the 4th International Workshop on Java Technologies for Real-Time and Embedded Systems (JTRES)*, ACM, pages 170–177, 2006.

AHMED, Z.; ZAHOOR, M.; YOUNAS, I. Mutation Operators for Object-Oriented Systems: A Survey. In: *The 2nd International Conference on Computer and Automation Engineering (ICCAE)*, pages 614–618, 2010.

AHO, A. V.; LAM, M. S.; SETHI, R.; ULLMAN, J. D. *Compilers: Principles, Techniques, & Tools.* 2nd edition. Addison Wesley, 1009 pages, 2007.

AL-IADAN, M. A Survey and a Taxonomy of Approaches for Testing Parallel and Distributed Programs. In: *ACS/IEEE International Conference on Computer Systems and Applications*, pages 273–279, 2001.

ALLMAN, E. A Conversation with James Gosling. *Queue*, volume 2, pages 24–33, 2004.

AMMANN, P.; OFFUTT, J. *Introduction to Software Testing.* Cambridge University Press, 344 pages, 2008.

ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys (CSUR)*, volume 15, number 1, pages 3–43,

1983.

Andrews, J. H.; Briand, L. C.; Labiche, Y. Is Mutation an Appropriate Tool for Testing Experiments? In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, ACM, pages 402–411, 2005.

Armbruster, A.; Baker, J.; Cunei, A.; Flack, C.; Holmes, D.; Pizlo, F.; Pla, E.; Prochazka, M.; Vitek, J. A Real-Time Java Virtual Machine With Applications in Avionics. *ACM Transactions on Embedded Computing Systems (TECS)*, volume 7, number 1, pages 5:1–5:49, 2007.

Arnold, K.; Gosling, J.; Holmes, D. *The Java Programming Language.* 4th edition. Prentice Hall, 928 pages, 2005a.

Arnold, M.; Fink, S.; Grove, D.; Hind, M.; Sweeney, P. A Survey of Adaptive Optimization in Virtual Machines. *Proceedings of the IEEE*, volume 93, number 2, pages 449–466, 2005b.

Aslam, F.; Schindelhauer, C.; Ernst, G.; Spyra, D.; Meyer, J.; Zalloom, M. Introducing TakaTuka: a Java Virtual Machine for Motes. In: *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems*, ACM, pages 399–400, 2008.

Aycock, J. A Brief History of Just-in-time. *ACM Computing Surveys (CSUR)*, volume 35, number 2, pages 97–113, 2003.

Azevedo, A.; Kejariwal, A.; Veidenbaum, A.; Nicolau, A. High Performance Annotation-aware JVM for Java Cards. In: *Proceedings of the 5th ACM International Conference on Embedded Software*, ACM, pages 52–61, 2005.

Baier, C.; Katoen, J.-P. *Principles of Model Checking.* The MIT Press, 984 pages, 2008.

Baker, J.; Cunei, A.; Flack, C.; Pizlo, F.; Prochazka, M.; Vitek, J.; Armbruster, A.; Pla, E.; Holmes, D. A Real-time Java Virtual Machine for Avionics – An Experience Report. In: *Proceedings of the 12th IEEE Symposium on Real-time and Embedded Technology and Applications*, IEEE, pages 384–396, 2006.

Ball, T.; Burckhardt, S.; de Halleux, P.; Musuvathi, M.; Qadeer, S. Predictable and Progressive Testing of Multithreaded Code. *IEEE Software*, volume 28, number 3, pages 75–83, 2011.

Ball, T.; Majumdar, R.; Millstein, T.; Rajamani, S. K. Automatic Predicate Abstraction of C Programs. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 36, number 5, pages 203–213, 2001.

Ballarin, E.; Burkhart, H.; Eigenmann, R.; Kindlimann, H.; Moser, M. Making a Compiler Easily Portable. *IEEE Software*, volume 5, number 3, pages 30–38,

1988.

Barbosa, E. F.; Maldonado, J. C.; Vincenzi, A. M. R. Toward the Determination of Sufficient Mutant Operators for C. *Software Testing, Verification and Reliability*, volume 11, number 2, pages 113–136, 2001.

Barham, P.; Dragovic, B.; Fraser, K.; Hand, S.; Harris, T.; Ho, A.; Neugebauer, R.; Pratt, I.; Warfield, A. Xen and the Art of Virtualization. *SIGOPS Operating System Reviews*, volume 37, pages 164–177, 2003.

Baruch, O.; Katz, S. Partially Interpreted Schemas for CSP Programming. *Science of Computer Programming*, volume 10, number 1, pages 1–18, 1988.

Bebenita, M.; Chang, M.; Wagner, G.; Gal, A.; Wimmer, C.; Franz, M. Trace-based Compilation in Execution Environments without Interpreters. In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*, ACM, pages 59–68, 2010.

Ben-Ari, M. *Principles of Concurrent and Distributed Programming.* 2nd edition. Addison Wesley, 384 pages, 2006.

Ben-Ari, M. M. A Primer on Model Checking. *ACM Inroads*, volume 1, number 1, pages 40–47, 2010.

Ben-Asher, Y.; Eytani, Y.; Farchi, E.; Ur, S. Noise Makers Need to Know Where to be Silent – Producing Schedules That Find Bugs. In: *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 458–465, 2006.

Berekovic, M.; Kloos, H.; Pirsch, P. Hardware Realization of a Java Virtual Machine for High Performance Multimedia Applications. In: *IEEE Workshop on Signal Processing Systems - Design and Implementation*, pages 479–488, 1997.

Bergan, T.; Anderson, O.; Devietti, J.; Ceze, L.; Grossman, D. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 45, number 3, pages 53–64, 2010.

Bloch, J. *Effective Java: Programming Language Guide.* 1st edition. Addison-Wesley, 272 pages, 2001.

Blum, A. L.; Langley, P. Selection of Relevant Features and Examples in Machine Learning. *Artificial Intelligence*, volume 97, number 1-2, pages 245–271, 1997.

Bocchino, Jr., R. L.; Adve, V. S.; Dig, D.; Adve, S. V.; Heumann, S.; Komuravelli, R.; Overbey, J.; Simmons, P.; Sung, H.; Vakilian, M. A Type and Effect System for Deterministic Parallel Java. *ACM Special Interest Group on*

Programming Languages (SIGPLAN) Notices, volume 44, number 10, pages 97–116, 2009.

Bogacki, B.; Walter, B. Evaluation of Test Code Quality with Aspect-Oriented Mutations. In: *Extreme Programming and Agile Processes in Software Engineering*, volume 4044 of *Lecture Notes in Computer Science*, Springer Berlin/Heidelberg, pages 202–204, 2006.

Bogacki, B.; Walter, B. Aspect-oriented Response Injection: An Alternative to Classical Mutation Testing. In: *Software Engineering Techniques: Design for Quality*, volume 227 of *International Federation for Information Processing (IFIP)*, Springer, pages 273–282, 2007.

Booth, W. C.; Colomb, G. G.; Williams, J. M. *The Craft of Research (Chicago Guides to Writing, Editing, and Publishing).* 3rd edition. University Of Chicago Press, 317 pages, 2008.

Boyapati, C.; Rinard, M. A Parameterized Type System for Race-Free Java Programs. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 36, number 11, pages 56–69, 2001.

Bradbury, J.; Cordy, J.; Dingel, J. ExMAn: A Generic and Customizable Framework for Experimental Mutation Analysis. In: *Second Workshop on Mutation Analysis*, pages 4–10, 2006a.

Bradbury, J.; Cordy, J.; Dingel, J. Mutation Operators for Concurrent Java (J2SE 5.0). In: *Second Workshop on Mutation Analysis*, pages 11–20, 2006b.

Bradbury, J. S.; Cordy, J. R.; Dingel, J. Comparative Assessment of Testing and Model Checking Using Program Mutation. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 210–222, 2007.

Brandner, F.; Thorn, T.; Schoeberl, M. Embedded JIT Compilation with CACAO on YARI. In: *IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC)*, pages 63–70, 2009.

Brilliant, S. S.; Wiseman, T. R. The First Programming Paradigm and Language Dilemma. In: *Proceedings of the Twenty-Seventh Special Interest Group on Computer Science Education (SIGCSE) Technical Symposium on Computer Science Education*, ACM, pages 338–342, 1996.

Brouwers, N.; Corke, P.; Langendoen, K. Darjeeling, A Java Compatible Virtual Machine for Microcontrollers. In: *Proceedings of the ACM/IFIP/USENIX Middleware Conference Companion*, ACM, pages 18–23, 2008.

Budd, T. A. *Mutation Analysis of Program Test Data.* Ph.D. Thesis, Yale University, 1980.

Budd, T. A.; Angluin, D. Two Notions of Correctness and Their Relation to Testing. *Acta Informatica*, volume 18, pages 31–45, 1982.

Budd, T. A.; Lipton, R. J.; DeMillo, R.; Sayward, F. The Design of a Prototype Mutation System for Program Testing. *International Workshop on Managing Requirements Knowledge*, pages 623–629, 1978.

Butenhof, D. R. *Programming with POSIX Threads.* Addison-Wesley Professional, 397 pages, 1996.

Byoungju, C.; Mathur, A. P. High-Performance Mutation Testing. *Journal of Systems and Software*, volume 20, number 2, pages 135–152, 1993.

Cantrill, B.; Bonwick, J. Real-World Concurrency. *Queue*, volume 6, number 5, pages 16–25, 2008a.

Cantrill, B.; Bonwick, J. Real-World Concurrency. *Communications of the ACM*, volume 51, number 11, pages 34–39, 2008b.

Carbone, M.; Zamboni, D.; Lee, W. Taming Virtualization. *IEEE Security and Privacy*, volume 6, pages 65–67, 2008.

Carver, R.; Tai, K. C. Replay and Testing for Concurrent Programs. *IEEE Software*, volume 8, number 2, pages 66–74, 1991.

Cavanagh, S.; Wang, Y. Design of a Real-time Virtual Machine (RTVM). In: *Canadian Conference on Electrical and Computer Engineering*, IEEE, pages 2021–2024, 2005.

Cesarini, F.; Thompson, S. *Erlang Programming.* O'Reilly Media, Inc., 496 pages, 2009.

Chen, G.; Kandemir, M. Improving Java Virtual Machine Reliability for Memory-constrained Embedded Systems. In: *Proceedings of the 42nd annual Design Automation Conference (DAC)*, ACM, pages 690–695, 2005.

Chen, J.; MacDonald, S. Testing Concurrent Programs Using Value Schedules. In: *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE)*, ACM, pages 313–322, 2007.

Chevalley, P.; Thévenod-Fosse, P. A Mutation Analysis Tool for Java Programs. *International Journal on Software Tools for Technology Transfer*, volume 5, number 1, pages 90–103, 2003.

Chilakamarri, K.-R.; Elbaum, S. Reducing Coverage Collection Overhead with Disposable Instrumentation. In: *15th International Symposium on Software Reliability*

Engineering (ISSRE), pages 233–244, 2004.

Choi, J.-D.; Alpern, B.; Ngo, T.; Sridharan, M.; Vlissides, J. A Perturbation-free Replay Platform for Cross-optimized Multithreaded Applications. In: *Proceedings of the 15th International Parallel and Distributed Processing Symposium*, IEEE Computer Society, pages 23–33, 2001.

Choi, J.-D.; Lee, K.; Loginov, A.; O'Callahan, R.; Sarkar, V.; Sridharan, M. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 37, number 5, pages 258–269, 2002.

Clarke, E. M.; Emerson, E. A.; Sifakis, J. Model Checking: Algorithmic Verification and Debugging. *Communications of the ACM*, volume 52, number 11, pages 74–84, 2009.

Clarke, E. M.; Wing, J. M. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys (CSUR) - Special ACM 50th-anniversary issue: Strategic Directions in Computing Research*, volume 28, number 4, pages 626–643, 1996.

Coffman, E. G.; Elphick, M.; Shoshani, A. System Deadlocks. *ACM Computing Surveys*, volume 3, number 2, pages 67–78, 1971.

Colburn, T.; Shute, G. Abstraction in Computer Science. *Minds and Machines*, volume 17, number 2, pages 169–184, 2007.

Corbett, J. C.; Dwyer, M. B.; Hatcliff, J.; Laubach, S.; Păsăreanu, C. S.; Robby; Zheng, H. Bandera: Extracting Finite-State Models from Java Source Code. In: *Proceedings of the 22nd International Conference on Software Engineering (ICSE)*, ACM, pages 439–448, 2000.

Craig, I. D. *Virtual Machines.* Springer, 269 pages, 2005.

Creasy, R. J. The Origin of the VM/370 Time-Sharing System. *IBM Journal of Research and Development*, volume 25, number 5, pages 483–490, 1981.

Dahl, O.-J.; Myhrhaug, B.; Nygaard, K. Some features of the SIMULA 67 language. In: *Proceedings of the Second Conference on Applications of Simulations*, Winter Simulation Conference, pages 29–31, 1968.

Davis, B.; Waldron, J. A Survey of Optimisations for the Java Virtual Machine. In: *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ)*, Computer Science Press, Inc., pages 181–183, 2003.

Deitel, H.; Deitel, P. *Java How to Program.* 6th edition. Prentice Hall, 1568 pages, 2004.

Delamare, R.; Baudry, B.; Le-Traon, Y. AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors. In: *International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 200–204, 2009.

Delamaro, M. E. Using Instrumentation to Reproduce the Execution of Java Concurrent Programs. In: *Proceedings of the Brazilian Symposium on Software Quality*, Brazilian Computer Society (SBC), 2004.

Delamaro, M. E.; Maldonado, J. C. Proteum - A Tool for Assessment of Test Adequacy for C Programs. In: *Proceedings of the Conference on Performability in Computing Systems*, pages 79–95, 1996.

Delamaro, M. E.; Pezzè, M.; Vincenzi, A. M. R.; Maldonado, J. C. Mutant Operators for Testing Concurrent Java Programs. In: *XV Simpósio Brasileiro de Engenharia de Software (SBES)*, pages 272–285, 2001.

DeMillo, R.; Guindi, D.; McCracken, W.; Offutt, A.; King, K. An Extended Overview of the Mothra Software Testing Environment. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 142–151, 1988.

DeMillo, R.; Lipton, R.; Sayward, F. Hints on Test Data Selection: Help for the Practicing Programmer. *Computer*, volume 11, pages 34–41, 1978.

Demillo, R. A.; Krauser, E.; Mathur, A. Compiler-Integrated Program Mutation. In: *Proceedings of the Fifteenth Annual International Computer Software and Applications Conference (COMPSAC)*, pages 351–356, 1991.

Deng, L.; Offutt, J.; Li, N. Empirical Evaluation of the Statement Deletion Mutation Operator. In: *6th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2013.

Derezinska, A.; Szustek, A. Tool-Supported Advanced Mutation Approach for Verification of C# Programs. In: *Third International Conference on Dependability of Computer Systems*, pages 261–268, 2008.

DeSouza, J.; Kuhn, B.; de Supinski, B. R.; Samofalov, V.; Zheltov, S.; Bratanov, S. Automated, Scalable Debugging of MPI Programs With Intel Message Checker. In: *Proceedings of the Second International Workshop on Software Engineering for High Performance Computing System Applications*, ACM, pages 78–82, 2005.

Deutsch, L. P.; Schiffman, A. M. Efficient Implementation of the Smalltalk-80 System. In: *Proceedings of the 11th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, ACM, pages 297–302, 1984.

D'Hondt, T. Are Bytecodes an Atavism? In: *First Workshop on Self-Sustaining Systems*, Lecture Notes in Computer Science, Springer Berlin/Heidelberg, pages 140–155,

2008.

Diehl, S. A Formal Introduction to the Compilation of Java. *Software: Practice and Experience*, volume 28, number 3, pages 297–327, 1998.

Dijkstra, E. Hierarchical Ordering of Sequential Processes. *Acta Informatica*, volume 1, number 2, pages 115–138, 1971.

Dijkstra, E. W. Solution of a Problem in Concurrent Programming Control. *Communications of the ACM*, volume 8, number 9, pages 569–569, 1965.

Dijkstra, E. W. *The Origin of Concurrent Programming*, chapter: Cooperating Sequential Processes Springer-Verlag, pages 65–138, 2002.

Do, H.; Rothermel, G. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering*, volume 32, number 9, pages 733–752, 2006.

Dodge, Y. *The Concise Encyclopedia of Statistics (Springer Reference).* 1 edition. Springer, 626 pages, 2009.

Downey, T. *Web Development with Java: Using Hibernate, JSPs and Servlets.* Springer, 302 pages, 2007.

Durelli, V. H. S.; Felizardo, K. R.; Delamaro, M. E. Systematic Mapping Study on High-level Language Virtual Machines (VMIL). In: *Virtual Machines and Intermediate Languages*, ACM, pages 1–6, 2010.

Eckel, B. *Thinking in Java.* 4th edition. Prentice Hall, 1150 pages, 2006.

Edelson, J.; Liu, H. *JRuby Cookbook.* O'Reilly Media, Inc., 224 pages, 2008.

Edelstein, O.; Farchi, E.; Goldin, E.; Nir, Y.; Ratsaby, G.; Ur, S. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, volume 15, number 3–5, pages 485–499, 2003.

Edelstein, O.; Farchi, E.; Nir, Y.; Ratsaby, G.; Ur, S. Multithreaded Java Program Test Generation. *IBM Systems Journal*, volume 41, number 1, pages 111–125, 2002.

Elsworth, E. F. Compilation Via an Intermediate Language. *The Computer Journal*, volume 22, number 3, pages 226–233, 1979.

Engel, J. *Programming for the Java Virtual Machine.* Addison-Wesley Professional, 512 pages, 1999.

Evans, B. J.; Verburg, M. *The Well-Grounded Java Developer: Vital Techniques of Java 7 and Polyglot Programming.* Manning Publications, 496 pages, 2012.

Eytani, Y.; Farchi, E.; Ben-Asher, Y. Heuristics for Finding Concurrent Bugs. In: *Proceedings of the International Parallel and Distributed Processing Symposium*, pages 8–15, 2003.

Eytani, Y.; Havelund, K.; Stoller, S. D.; Ur, S. Towards a Framework and a Benchmark for Testing Tools for Multi-Threaded Programs. *Concurrency and Computation: Practice and Experience*, volume 19, number 3, pages 267–279, 2007.

Eytani, Y.; Latvala, T. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. In: Bin, E.; Ziv, A.; Ur, S., *Hardware and Software, Verification and Testing*, volume 4383 of *Lecture Notes in Computer Science*, Springer, pages 183–197, 2007.

Farchi, E.; Nir, Y.; Ur, S. Concurrent Bug Patterns and How to Test Them. In: *Proceedings from the International Parallel and Distributed Processing Symposium*, pages 7–13, 2003.

Ferrari, F. C.; Nakagawa, E. Y.; Maldonado, J. C.; Rashid, A. Proteum/AJ: A Mutation System for AspectJ Programs. In: *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD)*, ACM, pages 73–74, 2011.

Flanagan, C.; Freund, S. N. Type-based Race Detection for Java. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 35, number 5, pages 219–232, 2000.

Flanagan, C.; Freund, S. N. Detecting Race Conditions in Large Programs. In: *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program Analysis for Software Tools and Engineering (PASTE)*, ACM, pages 90–96, 2001.

Flanagan, D.; Matsumoto, Y. *The Ruby Programming Language.* O'Reilly Media, Inc., 448 pages, 2008.

Fleyshgakker, V. N.; Weiss, S. N. Efficient Mutation Analysis: A New Approach. In: *Proceedings of the 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, ACM, pages 185–195, 1994.

Fong, A. S.; Yau, C.; Liu, Y. A Hardware-Software Integrated Design for a High-Performance Java Processor. *Third International Conference on Information Technology: New Generations*, pages 516–521, 2012.

Frankl, P. G.; Weiss, S. N.; Hu, C. All-uses vs Mutation Testing: An Experimental Comparison of Effectiveness. *Journal of Systems and Software*, volume 38, number 3, pages 235–253, 1997.

Fraser, S. D.; Gosling, J.; Hejlsberg, A.; Madsen, O. L.; Meyer, B.; Steele, G. Celebrating 40 Years of Language Evolution: Simula 67 to the Present and Beyond. In: *Companion to the 22nd annual ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Object-Oriented Programing, Systems, Languages,* *and Applications (OOPSLA)*, ACM, pages 1021–1023, 2007.

Gabbrielli, M.; Martini, S. *Programming Languages: Principles and Paradigms (Undergraduate Topics in Computer Science).* Springer, 459 pages, 2010.

Garg, V. K. *Concurrent and Distributed Computing in Java.* Wiley-IEEE Press, 336 pages, 2004.

Gatlin, K. S. Trials and Tribulations of Debugging Concurrency. *Queue*, volume 2, number 7, pages 66–73, 2004.

Girgis, M. R.; Woodward, M. R. An Integrated System for Program Testing Using Weak Mutation and Data Flow Analysis. In: *Proceedings of the 8th International Conference on Software Engineering (ICSE)*, IEEE, pages 313–319, 1985.

Gligoric, M.; Badame, S.; Johnson, R. SMutant: A Tool for Type-Sensitive Mutation Testing in a Dynamic Language. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ACM, pages 424–427, 2011.

Gligoric, M.; Jagannath, V.; Luo, Q.; Marinov, D. Efficient Mutation Testing of Multithreaded Code. *Software Testing, Verification and Reliability*, volume 23, number 5, pages 375–403, 2013a.

Gligoric, M.; Zhang, L.; Pereira, C.; Pokam, G. Selective Mutation Testing for Concurrent Code. In: *Proceedings of the 2013 International Symposium on Software Testing and Analysis (ISSTA)*, ACM, pages 224–234, 2013b.

Goetz, B.; Peierls, T.; Bloch, J.; Bowbeer, J.; Holmes, D.; Lea, D. *Java Concurrency in Practice.* Addison-Wesley Professional, 384 pages, 2006.

Goldberg, A.; Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley Longman Publishing Co., Inc., 714 pages, 1983.

Goldberg, R. P. Survey of Virtual Machine Research. *Computer*, pages 34–45, 1974.

Goosen, L. A Brief History of Choosing First Programming Languages. In: *History of Computing and Education*, volume 269 of *IFIP International Federation for Information Processing*, Springer Boston, pages 167–170, 2008.

Gosling, J. Java Intermediate Bytecodes: ACM Special Interest Group on Programming Languages Workshop on Intermediate Representations (IR). *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 30, number 3, pages 111–118, 1995.

Gough, J. K. Stacking Them up: a Comparison of Virtual Machines. In: *Proceedings of the 6th Australasian Computer Systems Architecture Conference (ACSAC)*, IEEE, pages 55–61, 2001.

GOUGH, J. K. Virtual Machines, Managed Code and Component Technology. In: *Australian Software Engineering Conference*, IEEE, pages 5–12, 2005.

GOUGH, K. J.; CORNEY, D. Evaluating the Java Virtual Machine as a Target for Languages Other Than Java. In: *Modular Programming Languages*, volume 1897 of *Lecture Notes in Computer Science*, Springer-Berlin, pages 278–290, 2000.

GREGG, D.; BEATTY, A.; CASEY, K.; DAVIS, B.; NISBET, A. The Case for Virtual Register Machines. *Science of Computer Programming*, volume 57, number 3, pages 319–338, 2005.

GRÖTKER, T.; HOLTMANN, U.; KEDING, H.; WLOKA, M. *The Developer's Guide to Debugging.* CreateSpace Independent Publishing Platform, 242 pages, 2012.

GROVES, L. J.; ROGERS, W. J. The Design of a Virtual Machine for Ada. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 15, pages 223–234, 1980.

GRÜN, B. J. M.; SCHULER, D.; ZELLER, A. The Impact of Equivalent Mutants. In: *Proceedings of the IEEE International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, IEEE, pages 192–199, 2009.

GUTHERY, S. Java Card: Internet Computing on a Smart Card. *IEEE Internet Computing*, volume 1, number 1, pages 57–59, 1997.

HAASE, C.; GUY, R. *Filthy Rich Clients: Developing Animated and Graphical Effects for Desktop Java Applications.* Prentice Hall, 608 pages, 2007.

HAFEEZ, Y.; ABBAS, M.; MUSTAFA, G. Techniques for Data-race Detection and Fault Tolerance: A Survey. In: *International Conference on Computer Information Science (ICCIS)*, pages 958–961, 2012.

HAINES, S.; POTTS, S. *Java 2 Primer Plus.* Sams, 840 pages, 2003.

HALEWOOD, K.; WOODWARD, M. A Uniform Graphical View of the Program Construction Process: GRIPSE. *International Journal of Man-Machine Studies*, volume 38, number 5, pages 805–837, 1993.

HAMLET, R. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, volume SE-3, number 4, pages 279–290, 1977.

HAND, S. An Experiment in Determinism. *Communications of the ACM*, volume 55, number 5, pages 110–110, 2012.

HANSEN, P. B. *Operating System Principles.* Prentice-Hall, Inc., 366 pages, 1973.

HANSEN, P. B. *History of Programming Languages – II*, chapter: Monitors and Concurrent Pascal: A Personal History ACM, pages 121–172, 1996.

Hardin, D. Crafting a Java Virtual Machine in Silicon. *IEEE Instrumentation Measurement Magazine*, volume 4, number 1, pages 54–56, 2001.

Harrold, M. J. Testing: A Roadmap. In: *Proceedings of the Conference on The Future of Software Engineering*, ACM, pages 61–72, 2000.

Havelund, K.; Pressburger, T. Model Checking Java Programs Using Java PathFinder. *International Journal on Software Tools for Technology Transfer*, volume 2, number 4, pages 366–381, 2000.

Hennessy, J. L.; Patterson, D. A. *Computer Architecture: A Quantitative Approach.* 4th edition. Morgan Kaufmann, 8–12 pages, 2006.

Higuera-Toledano, M.; Issarny, V.; Banatre, M.; Cabillic, G.; Lesot, J. P.; Parain, F. Java Embedded Real-time Systems: An Overview of Existing Solutions. In: *Proceedings of the third IEEE International Symposium on Object-Oriented Real-time Distributed Computing (ISORC)*, IEEE, pages 392–399, 2000.

Higuera-Toledano, M. T. About 15 Years of Real-time Java. In: *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES)*, ACM, pages 34–43, 2012.

Hoare, C. A. R. Monitors: an Operating System Structuring Concept. *Communications of the ACM*, volume 17, number 10, pages 549–557, 1974.

Holloman, E. D. *Design and Implementation of a Replay Debugger for Parallel Programs on Unix-Based Systems.* Master thesis, Computer Science Department, 1989.

Holzmann, G. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, volume 23, number 5, pages 279–295, 1997.

Horgan, J. R.; Mathur, A. P. *Weak Mutation is Probably Strong Mutation.* Technical Report SERC-TR-83-P, Purdue University, 1990.

Howden, W. Weak Mutation Testing and Completeness of Test Sets. *IEEE Transactions on Software Engineering*, volume 8, number 4, pages 371–379, 1982.

Hower, D.; Dudnik, P.; Hill, M.; Wood, D. Calvin: Deterministic or Not? Free Will to Choose. In: *IEEE 17th International Symposium on High Performance Computer Architecture (HPCA)*, pages 333–334, 2011.

Hu, J.; Li, N.; Offutt, J. An analysis of OO mutation operators. In: *Seventh Workshop on Mutation Analysis (IEEE Mutation 2011)*, Berlin, Germany, pages 334–341, 2011.

Husaini, S. F. Using the Java Native Interface. *Crossroads*, volume 4, number 2, pages 18–23, 1997.

Hussain, S. *Mutation Clustering.* Ph.D. Thesis, King's College London, 2008.

Irvine, S.; Pavlinic, T.; Trigg, L.; Cleary, J.; Inglis, S.; Utting, M. Jumble Java Bytecode to Measure the Effectiveness of Unit Tests. In: *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION*, pages 169–175, 2007.

Javier, F. *Java 7 Concurrency Cookbook.* Packt Publishing, 364 pages, 2012.

Ji, C.; Chen, Z.; Xu, B.; Zhao, Z. A Novel Method of Mutation Clustering Based on Domain Analysis. In: *Proceedings of the 21st International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 422–425, 2009.

Jia, Y.; Harman, M. MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language. In: *Testing: Academic Industrial Conference, Practice and Research Techniques (TAIC PART).*, pages 94–98, 2008.

Jia, Y.; Harman, M. Higher Order Mutation Testing. *Information and Software Technology*, volume 51, number 10, pages 1379–1393, 2009.

Jia, Y.; Harman, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering*, volume 37, number 5, pages 649–678, 2011.

Jikes RVM Project Jikes Research Virtual Machine (RVM) Project. Available at: `http://jikesrvm.org/`, Accessed August 15, 2013.

Johnson, R. A.; Bhattacharyya, G. K. *Statistics: Principles and Methods.* John Wiley & Sons, 704 pages, 2009.

Joisha, P. G.; Midkiff, S. P.; Serrano, M. J.; Gupta, M. Efficiently Adapting Java Binaries in Limited Memory Contexts. *International Journal of Parallel Programming*, volume 30, number 4, pages 257–289, 2002.

Jones, R.; Hosking, A.; Moss, E. *The Garbage Collection Handbook: The Art of Automatic Memory Management.* Chapman and Hall/CRC, 511 pages, 2011.

Joshi, P.; Naik, M.; Park, C.; Sen, K. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs. In: *Proceedings of the 21st International Conference on Computer Aided Verification (CAV)*, Springer, pages 675–681, 2009.

Kapfhammer, G. M.; Soffa, M. L.; Mosse, D. Testing in Resource Constrained Execution Environments. In: *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering (ASE)*, ACM, pages 418–422, 2005.

Kapoor, K. Formal Analysis of Coupling Hypothesis for Logical Faults. *Innovations in Systems and Software Engineering*, volume 2, pages 80–87, 2006.

Kay, A. C. The Early History of Smalltalk. In: *History of Programming Languages – II*, ACM, pages 511–598, 1996.

KAZI, I. H.; CHEN, H. H.; STANLEY, B.; LILJA, D. J. Techniques for Obtaining High Performance in Java Programs. *ACM Computing Surveys (CSUR)*, volume 32, number 3, pages 213–240, 2000.

KELBLEY, J.; STERLING, M. *Windows Server 2008 R2 Hyper-v: Insiders Guide to Microsoft's Hypervisor.* 2nd edition. Sybex, 408 pages, 2008.

KERNIGHAN, B. W.; RITCHIE, D. M. *C Programming Language.* 2nd edition. Prentice Hall, 274 pages, 1988.

KILLALEA, T. Meet the Virts. *ACM Queue*, volume 6, pages 14–18, 2008.

KIM, H.; BOND, R. Multicore Software Technologies. *IEEE Signal Processing Magazine*, volume 26, number 6, pages 80–89, 2009.

KING, K. N.; OFFUTT, J. A Fortran Language System for Mutation-based Software Testing. *Software-Practice and Experience*, volume 21, number 7, pages 685–718, 1991.

KLINT, P. Interpretation Techniques. *Software: Practice and Experience*, volume 11, number 9, pages 963–973, 1981.

KOENIG, D.; GLOVER, A.; KING, P.; LAFORGE, G.; SKEET, J. *Groovy in Action.* Manning Publications, 696 pages, 2007.

KOHAVI, R.; JOHN, G. H. Wrappers for Feature Subset Selection. *Artificial Intelligence*, volume 97, number 1-2, pages 273–324, 1997.

KOSHY, J.; PANDEY, R. VMSTAR: Synthesizing Scalable Runtime Environments for Sensor Networks. In: *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, ACM, pages 243–254, 2005.

KOSHY, J.; PANDEY, R.; WIRJAWAN, I. Optimizing Embedded Virtual Machines. In: *International Conference on Computational Science and Engineering (CSE)*, IEEE, pages 342–351, 2009.

KUROTSUCHI, B. T. The Wonders of Java Object Serialization. *Crossroads*, volume 4, number 2, pages 3–8, 1997.

LANGDON, W. B.; HARMAN, M.; JIA, Y. Multi Objective Higher Order Mutation Testing with Genetic Programming. In: *Proceedings of the 2009 Testing: Academic and Industrial Conference – Practice and Research Techniques*, IEEE, pages 21–29, 2009.

LANGDON, W. B.; HARMAN, M.; JIA, Y. Efficient Multi-Objective Higher Order Mutation Testing With Genetic Programming. *Journal of Systems and Software*, volume 83, number 12, pages 2416–2430, 2010.

LARSON, J. Erlang for Concurrent Programming. *Queue*, volume 6, number 5, pages 18–23, 2008.

Larson, J. Erlang for Concurrent Programming. *Communications of the ACM*, volume 52, number 3, pages 48–56, 2009.

Lawton, G. Moving Java Into Mobile Phones. *Computer*, volume 35, number 6, pages 17–20, 2002.

Lee, A. H.; Zachary, J. L. Reflections on Metaprogramming. *IEEE Transactions on Software Engineering*, volume 21, pages 883–893, 1995.

Lee, E. A. The Problem with Threads. *Computer*, volume 39, number 5, pages 33–42, 2006.

Levis, P.; Culler, D. Maté: A Tiny Virtual Machine for Sensor Networks. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 37, number 10, pages 85–95, 2002.

Li, N.; Praphamontripong, U.; Offutt, J. An Experimental Comparison of Four Unit Test Criteria: Mutation, Edge-Pair, All-uses and Prime Path Coverage. In: *Fifth Workshop on Mutation Analysis (IEEE Mutation 2009)*, pages 220–229, 2009.

Li, Q. Java Virtual Machine – Present and Near Future. *International Conference on Technology of Object-Oriented Languages*, page 480, 1998.

Liang, S. *The Java Native Interface: Programmer's Guide and Specification (The Java Series)*. Addison-Wesley Professiona, 320 pages, 1999.

Lindholm, T.; Yellin, F. *The Java Virtual Machine Specification.* 2nd edition. Prentice Hall, 496 pages, 1999.

Lindholm, T.; Yellin, F.; Bracha, G.; Buckley, A. The Java™ Virtual Machine Specification Java SE 7 Edition. Available at: `http://docs.oracle.com/javase/specs/jvms/se7/jvms7.pdf`, Accessed June 11, 2012.

Lindsey, C. H. A History of Algol 68. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 28, number 3, pages 97–132, 1993.

Lipton, R. *Fault Diagnosis of Computer Programs.* Student report, Carnegie Mellon University, 1971.

Liu, T.; Curtsinger, C.; Berger, E. D. Dthreads: Efficient Deterministic Multithreading. In: *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP)*, ACM, pages 327–336, 2011.

Lo, C.-T.; Srisa-an, W.; Chang, J. Who Is Collecting Your Java Garbage? *IT Professional*, volume 5, number 2, pages 44–50, 2003.

Long, B.; Hoffman, D.; Strooper, P. Tool Support for Testing Concurrent Java Components. *IEEE Transactions on Software Engineering*, volume 29, number 6, pages 555–566, 2003.

Lu, H.-I.; Klein, P.; Netzer, R. Detecting Race Conditions in Parallel Programs that Use One Semaphore. In: *Algorithms and Data Structures*, volume 709 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pages 471–482, 1993.

Lu, S.; Park, S.; Seo, E.; Zhou, Y. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 43, number 3, pages 329–339, 2008.

Lutz, M. *Learning Python: Powerful Object-oriented Programming.* 4th edition. O'Reilly Media, Inc., 1216 pages, 2009.

Ma, Y. S.; Offutt, J.; Kwon, Y. R. MuJava : An Automated Class Mutation System. *Software Testing, Verification, and Reliability*, volume 15, number 2, pages 97–133, 2005.

Ma, Y.-S.; Offutt, J.; Kwon, Y.-R. MuJava: A Mutation System for Java. In: *Proceedings of the 28th International Conference on Software Engineering (ICSE)*, ACM, pages 827–830, 2006.

Madeyski, L.; Radyk, N. Judy – A Mutation Testing Tool for Java. *IET Software*, volume 4, number 1, pages 32–42, 2010.

Madiraju, P.; Namin, A. Paraμ – A Partial and Higher-Order Mutation Tool with Concurrency Operators. In: *IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 351–356, 2011.

Maldonado, J. C.; Delamaro, M. E.; Fabbri, S. C. P. F.; Simão, A. d. S.; Sugeta, T.; Vincenzi, A. M. R.; Masiero, P. C. Proteum: a Family of Tools to Support Specification and Program Testing Based on Mutation. In: *Mutation Testing for the New Century*, Kluwer Academic Publishers, pages 113–116, 2001.

Malkis, A.; Podelski, A.; Rybalchenko, A. Precise Thread-Modular Verification. In: *Static Analysis*, volume 4634 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pages 218–232, 2007.

Marick, B. The Weak Mutation Hypothesis. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, ACM, pages 190–199, 1991.

Marr, S.; Haupt, M.; D'Hondt, T. Intermediate Language Design of High-Level Language Virtual Machines: Towards Comprehensive Concurrency Support. In: *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages (VMIL)*, ACM, pages 3:1–3:2, 2009.

Mateo, P.; Usaola, M. Mutant Execution Cost Reduction: Through MUSIC (Mutant Schema Improved with Extra Code). In: *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, pages 664–672, 2012.

Mateo, P.; Usaola, M.; Offutt, J. Mutation at System and Functional Levels. In: *2010 Third International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, pages 110–119, 2010.

Mateo, P. R.; Usaola, M. P.; Offutt, J. Mutation at the Multi-class and System Levels. *Science of Computer Programming*, volume 78, number 4, pages 364–387, 2013.

Mathiske, B. Leveraging Meta-Circularity in the Maxine VM – The Maxine VM (Presentation). Available at: `http://wiki.jvmlangsummit.com/pdf/12_Mathiske_max.pdf`, JVM Language Summit, September 24, Santa Clara, California, Accessed March 6, 2013, 2008.

McDowell, C. E.; Helmbold, D. P. Debugging Concurrent Programs. *ACM Computing Surveys (CSUR)*, volume 21, number 4, pages 593–622, 1989.

McGhan, H.; O'Connor, M. PicoJava: A Direct Execution Engine For Java Bytecode. *Computer*, volume 31, number 10, pages 22–30, 1998.

Meijer, E.; Gough, J. K. Technical Overview of the Common Language Runtime. Available at: `http://research.microsoft.com/en-us/um/people/emeijer/papers/clr.pdf`, Accessed February 7, 2012.

Mickel, A. B. Pascal. In: *Encyclopedia of Computer Science*, 4th edition, John Wiley & Sons, pages 1372–1373, 2000.

Microsoft Windows Virtual PC. Available at `http://www.microsoft.com/windows/virtual-pc`, Accessed August 30, 2011.

Mogensen, T. Æ. *Introduction to Compiler Design (Undergraduate Topics in Computer Science).* Springer, 225 pages, 2011.

Moore, I. Jester - The JUnit Test Tester. Available at: `http://jester.sourceforge.net/`, Accessed May 24, 2013.

Morell, L. Theoretical Insights Into Fault-based Testing. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, IEEE, pages 45–62, 1988.

Morell, L. J. *A Theory of Error-based Testing.* Ph.D. Thesis, College Park, MD, USA, 1983.

Mresa, E. S.; Bottaci, L. Efficiency of Mutation Operators and Selective Mutation Strategies: An Empirical Study. *Software Testing, Verification and Reliability*, volume 9, number 4, pages 205–232, 1999.

Mughal, K.; Rasmussen, R. *A Programmer's Guide to Java SCJP Certification: A Comprehensive Primer.* 3rd edition. Addison-Wesley Professional, 1038 pages, 2008.

Naik, S.; Tripathy, P. *Software Testing and Quality Assurance: Theory and Practice.* Wiley-Spektrum, 648 pages, 2008.

Namin, A.; Andrews, J. Finding Sufficient Mutation Operators via Variable Reduction. In: *Second Workshop on Mutation Analysis*, 2006.

Namin, A. S.; Andrews, J. H. On Sufficiency of Mutants. In: *Companion to the Proceedings of the 29Th International Conference on Software Engineering*, IEEE, pages 73–74, 2007.

Namin, A. S.; Andrews, J. H.; Murdoch, D. J. Sufficient Mutation Operators for Measuring Test Effectiveness. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, ACM, pages 351–360, 2008.

Namin, A. S.; Kakarla, S. The Use of Mutation in Testing Experiments and its Sensitivity to External Threats. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA)*, ACM, pages 342–352, 2011.

Nelson, P. A. A Comparison of Pascal Intermediate Languages. In: *Proceedings of the 1979 ACM Special Interest Group on Programming Languages (SIGPLAN) Symposium on Compiler Construction*, ACM, pages 208–213, 1979.

Netzer, R. H. B.; Miller, B. P. Improving the Accuracy of Data Race Detection. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 26, number 7, pages 133–144, 1991.

Netzer, R. H. B.; Miller, B. P. What are Race Conditions?: Some Issues and Formalizations. *ACM Letters on Programming Languages and Systems (LOPLAS)*, volume 1, number 1, pages 74–88, 1992.

Nielson, F.; Nielson, H. Type and Effect Systems. In: *Correct System Design*, volume 1710 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pages 114–136, 1999.

Niemeyer, P.; Knudsen, J. *Learning Java.* O'Reilly Media, Inc., 984 pages, 2005.

Nutter, C. O.; Enebo, T.; Sieger, N.; Bini, O.; Dees, I. *Using JRuby: Bringing Ruby to Java.* O'Reilly Media, Inc., 300 pages, 2011.

O'Connor, J.; Tremblay, M. picoJava-I: the Java Virtual Machine in Hardware. *IEEE Micro*, volume 17, number 2, pages 45–53, 1997.

Odersky, M.; Spoon, L.; Venners, B. *Programming in Scala: A Comprehensive Step-by-Step Guide.* Artima Inc., 852 pages, 2011.

Oestreicher, M. Transactions in Java Card. In: *Proceedings of the 15th Annual Computer Security Applications Conference (ACSAC)*, pages 291–298, 1999.

Offutt, J. The Coupling Effect: Fact or Fiction. *Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes*, volume 14, number 8, pages 131–140, 1989.

Offutt, J. Investigations of the Software Testing Coupling Effect. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 1, number 1, pages 5–20, 1992.

Offutt, J. A Mutation Carol: Past, Present and Future. *Information and Software Technology*, volume 53, number 10, pages 1098–1107, Special Section on Mutation Testing, 2011.

Offutt, J.; King, K. N. A Fortran 77 Interpreter for Mutation Analysis. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 22, number 7, pages 177–188, 1987.

Offutt, J.; Lee, A.; Rothermel, G.; Untch, R.; Zapf, C. An Experimental Determination of Sufficient Mutation Operators. *ACM Transactions on Software Engineering Methodology*, volume 5, number 2, pages 99–118, 1996a.

Offutt, J.; Lee, S. D. How Strong Is Weak Mutation? In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, ACM, pages 200–213, 1991.

Offutt, J.; Lee, S. D. An Empirical Evaluation of Weak Mutation. *IEEE Transactions on Software Engineering*, volume 20, number 5, pages 337–344, 1994.

Offutt, J.; Ma, Y. S.; Kwon, Y. R. An Experimental Mutation System for Java. *Special Interest Group on Software Engineering (SIGSOFT) Software Engineering Notes*, volume 29, number 5, pages 1–4, 2004.

Offutt, J.; Rothermel, G.; Zapf, C. An Experimental Evaluation of Selective Mutation. In: *Proceedings of the 15th International Conference on Software Engineering (ICSE)*, IEEE, pages 100–107, 1993.

Offutt, J.; Untch, R. H. Mutation 2000: Uniting the Orthogonal. In: *Mutation Testing for the New Century*, Kluwer Academic Publishers, pages 34–44, 2001.

Offutt, J.; Voas, J.; Payne, J. *Mutation Operators for Ada.* Technical report, George Mason University, 1996b.

Olszewski, M.; Ansel, J.; Amarasinghe, S. Kendo: Efficient Deterministic Multithreading in Software. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 44, number 3, pages 97–108, 2009.

Oracle Corporation VirtualBox. Available at: `http://www.virtualbox.org/`, Accessed August 31, 2011.

ORACLE CORPORATION Code Eviction in the Maxine VM. Available at: `https://wikis.oracle.com/display/MaxineVM/Code+Eviction`, Accessed January 25, 2012.

ORACLE CORPORATION Maxine Virtual Machine Project. Available at: `http://wikis.sun.com/display/MaxineVM/Home`, Accessed June 11, 2013a.

ORACLE CORPORATION OpenJDK. Available at: `http://openjdk.java.net/`, Accessed July 20, 2013b.

ORACLE CORPORATION The Java HotSpot Performance Engine Architecture. Available at: `http://www.oracle.com/technetwork/java/whitepaper-135217.html`, Accessed June 20, 2013c.

OUSTERHOUT, J. Scripting: Higher Level Programming for the 21st Century. *Computer*, volume 31, number 3, pages 23–30, 1998.

PARALLELS HOLDINGS LTD. Parallels Desktop 4 for Windows & Linux. Available at `http://www.parallels.com/products/desktop`, Accessed August 30, 2011.

PARK, S.; LU, S.; ZHOU, Y. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 44, number 3, pages 25–36, 2009.

PARNAS, D. L. The Secret History of Information Hiding. In: *Software Pioneers*, Springer-Verlag, pages 399–409, 2002.

PEARS, A.; SEIDMAN, S.; MALMI, L.; MANNILA, L.; ADAMS, E.; BENNEDSEN, J.; DEVLIN, M.; PATERSON, J. A Survey of Literature on the Teaching of Introductory Programming. *ACM Special Interest Group on Computer Science Education (SIGCSE) Bulletin*, volume 39, pages 204–223, 2007.

PEDRONI, S.; RAPPIN, N. *Jython Essentials.* O'Reilly Media, Inc., 304 pages, 2002.

PIUMARTA, I.; RICCARDI, F. Optimizing Direct Threaded Code by Selective Inlining. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 33, number 5, pages 291–300, 1998.

PIZLO, F.; ZIAREK, L.; BLANTON, E.; MAJ, P.; VITEK, J. High-Level Programming of Embedded Hard Real-Time Devices. In: *Proceedings of the 5Th European Conference on Computer Systems (EuroSys)*, ACM, pages 69–82, 2010.

POPEK, G. J.; GOLDBERG, R. P. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, volume 17, number 7, pages 412–421, 1974.

PROKOPSKI, G. B.; VERBRUGGE, C. Compiler-Guaranteed Safety in Code-Copying Virtual Machines. In: *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, Springer-Verlag, pages 163–177, 2008.

PUFFITSCH, W.; SCHOEBERL, M. picoJava-II in an FPGA. In: *Proceedings of the 5th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM, pages 213–221, 2007.

RADHAKRISHNAN, R.; VIJAYKRISHNAN, N.; JOHN, L.; SIVASUBRAMANIAM, A.; RUBIO, J.; SABARINATHAN, J. Java Runtime Systems: Characterization and Architectural Implications. *IEEE Transactions on Computers*, volume 50, number 2, pages 131–146, 2001.

RAU, B. R. Levels of Representation of Programs and the Architecture of Universal Host Machines. In: *Proceedings of the 11th annual workshop on Microprogramming (MICRO)*, IEEE Press, pages 67–79, 1978.

RAYMOND, E. S. *The New Hacker's Dictionary.* 3rd edition. The MIT Press, 547 pages, 1996.

RAZA, A. A Review of Race Detection Mechanisms. In: *Computer Science – Theory and Applications*, volume 3967 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pages 534–543, 2006.

REEK, K. A. The Well-tempered Semaphore: Theme With Variations. *Proceedings of the Special Interest Group on Computer Science Education (SIGCSE) Bulletin - Inroads: Paving the Way Towards Excellence in Computing Education*, volume 34, number 1, pages 356–359, 2002.

RICKEN, M.; CARTWRIGHT, R. ConcJUnit: Unit Testing for Concurrent Programs. In: *Proceedings of the 7th International Conference on Principles and Practice of Programming in Java (PPPJ)*, ACM, pages 129–132, 2009.

RIGGS, R.; HUOPANIEMI, J.; TAIVALSAARI, A.; PATEL, M.; UOTILA, A. *Programming wireless devices with the java 2 platform, micro edition.* 2nd edition. Sun Microsystems, Inc., 2003.

RINARD, M. C.; LAM, M. S. The Design, Implementation, and Evaluation of Jade. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, volume 20, number 3, pages 483–545, 1998.

RITCHIE, D. M. The Development of the C Programming Language. In: *History of Programming Languages – II*, ACM, pages 671–698, 1996.

RONSSE, M.; DE BOSSCHERE, K. RecPlay: A Fully Integrated Practical Record/Replay System. *ACM Transactions on Computer Systems (TOCS)*, volume 17, number 2, pages 133–152, 1999.

Rosenblum, M. The Reincarnation of Virtual Machines. *ACM Queue*, volume 2, number 5, pages 34–40, 2004.

Russinovich, M.; Cogswell, B. Replay for Concurrent Non-deterministic Shared-memory Applications. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 31, number 5, pages 258–266, 1996.

Ryder, B. G.; Soffa, M. L. Influences on the Design of Exception Handling: ACM SIGSOFT Project on the Impact of Software Engineering Research on Programming Language Design. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 38, number 6, pages 16–22, 2003.

Ryder, B. G.; Soffa, M. L.; Burnett, M. The Impact of Software Engineering Research on Modern Programming Languages. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, volume 14, number 4, pages 431–477, 2005.

Sabharwal, C. Java, Java, Java. *IEEE Potentials*, volume 17, number 3, pages 33–37, 1998.

Sahinoglu, M.; Spafford, E. H. A Bayes Sequential Statistical Procedure for Approving Software Products. In: *Proceedings of the IFIP Conference on Approving Software Products (ASP)*, Elsevier, pages 43–56, 1990.

Sammet, J. E. Programming Languages: History and Future. *Communications of the ACM*, volume 15, number 7, pages 601–610, 1972.

Sammet, J. E. Why Ada Is Not Just Another Programming Language. *Communications of the ACM*, volume 29, number 8, pages 722–732, 1986.

Sandén, B. Coping with Java Threads. *Computer*, volume 37, number 4, pages 20–27, 2004.

Sasada, K. YARV: Yet Another RubyVM: Innovating the Ruby Interpreter. In: *Companion to the 20th annual ACM Special Interest Group on Programming Languages (SIGPLAN) Conference on Object-Oriented Programing, Systems, Languages, and Applications (OOPSLA)*, ACM, pages 158–159, 2005.

Savage, S.; Burrows, M.; Nelson, G.; Sobalvarro, P.; Anderson, T. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. *ACM Transactions on Computer Systems (TOCS)*, volume 15, number 4, pages 391–411, 1997.

Schoeberl, M. Hardware Support for Embedded Java. In: *Distributed, Embedded and Real-time Java Systems*, Springer, pages 159–176, 2012.

Schuler, D.; Dallmeier, V.; Zeller, A. Efficient mutation testing by checking invariant violations. In: *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA)*, ACM, pages 69–80, 2009.

Schuler, D.; Zeller, A. Javalanche: Efficient Mutation Testing for Java. In: *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, ACM, pages 297–298, 2009.

Schwartz, R. L.; brian d foy; Phoenix, T. *Learning Perl.* O'Reilly Media, Inc., 388 pages, 2011.

Scott, M. L. *Programming Language Pragmatics.* 3rd edition. Morgan Kaufmann, 944 pages, 2009.

Seawright, L. H.; MacKinnon, R. A. VM/370: A Study of Multiplicity and Usefulness. *IBM Systems Journal*, volume 18, number 1, pages 4–17, 1979.

Sebesta, R. W. *Concepts of Programming Languages.* 10th edition. Addison-Wesley, 816 pages, 2012.

Sen, K. Effective Random Testing of Concurrent Programs. In: *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ACM, pages 323–332, 2007.

Shi, Y.; Casey, K.; Ertl, M. A.; Gregg, D. Virtual Machine Showdown: Stack Versus Registers. *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 4, number 4, pages 2:1–2:36, 2008.

Sierra, K.; Bates, B. *SCJP Sun Certified Programmer for Java 6 Exam 310-065.* McGraw-Hill Osborne Media, 851 pages, 2008.

Silberschatz, A.; Galvin, P. B.; Gagne, G. *Operating System Concepts with Java.* 8th edition. Wiley, 1040 pages, 2009.

Silva, R. A. *Mutation Testing Applied to Concurrent MPI Programs (in Portuguese).* Master thesis, University of São Paulo, 2013.

Silva, R. A.; De Souza, S. d. R. S.; De Souza, P. S. L. Mutation Operators for Concurrent Programs in MPI. In: *Proceedings of the 13th Test Workshop Latin American (LATW)*, pages 1–6, 2012.

Simon, D.; Cifuentes, C. The Squawk Virtual Machine: Java on the Bare Metal. In: *Companion to the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, ACM, pages 150–151, 2005.

Simon, D.; Cifuentes, C.; Cleal, D.; Daniels, J.; White, D. Java on the Bare Metal of Wireless Sensor Devices: the Squawk Java Virtual Machine. In: *Proceedings of the 2nd International Conference on Virtual Execution Environments*, ACM, pages 78–88, 2006.

SINGER, J. JVM Versus CLR: A Comparative Study. In: *Proceedings of the 2nd International Conference on Principles and Practice of Programming in Java (PPPJ)*, ACM, pages 167–169, 2003.

SMITH, J.; NAIR, R. The Architecture of Virtual Machines. *Computer*, volume 38, number 5, pages 32–38, 2005a.

SMITH, J.; NAIR, R. *Virtual Machines: Versatile Platforms for Systems and Processes.* Morgan Kaufmann, 656 pages, 2005b.

SPINELLIS, D. Virtualize Me. *IEEE Software*, volume 29, number 5, pages 91–93, 2012.

SPRINKLE, J.; MERNIK, M.; TOLVANEN, J.; SPINELLIS, D. Guest Editors' Introduction: What Kinds of Nails Need a Domain-Specific Hammer? *IEEE Software*, volume 26, number 4, pages 15–18, 2009.

STALLINGS, W. *Operating Systems: Internals and Design Principles.* 7th edition. Prentice Hall, 816 pages, 2011.

STEELE, JR., G. L.; GABRIEL, R. P. The evolution of Lisp. *ACM Special Interest Group on Programming Languages (SIGPLAN) Notices*, volume 28, pages 231–270, 1993.

STOLLER, S. D. Testing Concurrent Java Programs using Randomized Scheduling. *Electronic Notes in Theoretical Computer Science*, volume 70, number 4, pages 142–157, 2002.

SUBRAMANIAM, V. *Programming Concurrency on the JVM: Mastering Synchronization, STM, and Actors.* Pragmatic Bookshelf, 280 pages, 2011.

SUTTER, H.; LARUS, J. Software and the Concurrency Revolution. *Queue*, volume 3, number 7, pages 54–62, 2005.

TAN, Y.; YAU, C.; LO, K.; YU, W.; MOK, P.; FONG, A. Design and Implementation of a Java Processor. *IEE Proceedings on Computers and Digital Techniques*, volume 153, number 1, pages 20–30, 2006.

TANENBAUM, A. S. *Modern Operating Systems.* 3rd edition. Prentice Hall, 1104 pages, 2007.

TEETOR, P. *R Cookbook.* O'Reilly, 436 pages, 2011.

THALMANN, N.; THALMANN, D. The Use of Pascal as a Teaching Tool in Introductory, Intermediate and Advanced Computer Science Courses. *ACM Special Interest Group on Computer Science Education (SIGCSE) Bulletin*, volume 10, pages 277–281, 1978.

THIES, W.; KARCZMAREK, M.; AMARASINGHE, S. StreamIt: A Language for Streaming Applications. In: *Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*, Springer Berlin Heidelberg, pages 179–196, 2002.

Thomas, D.; Fowler, C.; Hunt, A. *Programming Ruby 1.9: The Pragmatic Programmers' Guide.* 3rd edition. Pragmatic Bookshelf, 920 pages, 2009.

Thomm, I.; Stilkerich, M.; Wawersich, C.; Schröder-Preikschat, W. KESO: An Open-source Multi-JVM for Deeply Embedded Systems. In: *Proceedings of the 8th International Workshop on Java Technologies for Real-Time and Embedded Systems*, ACM, pages 109–119, 2010.

Till Harbaum The NanoVM – Java for the AVR. Available at: `http://www.harbaum.org/till/nanovm/index.shtml`, Accessed June 12, 2013.

Treese, W. Virtualization Virtually Everywhere. *Magazine netWorker*, volume 9, pages 13–15, 2005.

Untch, R. On Reduced Neighborhood Mutation Analysis Using a Single Mutagenic Operator. In: *ACM Southeast Regional Conference*, pages 19–21, 2009.

Untch, R. H. Mutation-based Software Testing Using Program Schemata. In: *Proceedings of the 30th Annual Southeast Regional Conference*, ACM, pages 285–291, 1992.

Untch, R. H.; Offutt, A. J.; Harrold, M. J. Mutation Analysis Using Mutant Schemata. *ACM SIGSOFT Software Engineering Notes*, volume 18, number 3, pages 139–148, 1993.

Usaola, M. P.; Mateo, P. R. Mutation Testing Cost Reduction Techniques: A Survey. *IEEE Software*, volume 27, number 3, pages 80–86, 2010.

Usaola, M. P.; Piattini, M.; García-Rodríguez, I. Decreasing the Cost of Mutation Testing with Second-order Mutants. *Software Testing, Verification and Reliability*, volume 19, number 2, pages 111–131, 2009.

Usaola, M. P.; Tendero, S.; Piattini, M. Integrating Techniques and Tools for Testing Automation. *Software Testing, Verification and Reliability*, volume 17, number 1, pages 3–39, 2007.

VMware Inc. VMware Server™. Available at `http://www.vmware.com/products/server/overview.html`, Accessed August 30, 2011.

Wah, K. S. H. T. Fault Coupling in Finite Bijective Functions. *Software Testing, Verification and Reliability*, volume 5, number 1, pages 3–47, 1995.

Wah, K. S. H. T. A Theoretical Study of Fault Coupling. *Software Testing, Verification and Reliability*, volume 10, number 1, pages 3–45, 2000.

Wah, K. S. H. T. An Analysis of the Coupling Effect I: Single Test Data. *Science of Computer Programming*, volume 48, number 2-3, pages 119–161, 2003.

Wald, A. Sequential Tests of Statistical Hypotheses. *The Annals of Mathematical Statistics*, volume 16, number 2, pages 117–186, 1945.

Waldo, J. *Java: The Good Parts.* O'Reilly Media, Inc., 194 pages, 2010.

Waldspurger, C. A. Memory Resource Management in VMware ESX Server. *SIGOPS Operating Systems Review*, volume 36, pages 181–194, 2002.

Wampler, D.; Payne, A. *Programming Scala: Scalability = Functional Programming + Objects.* O'Reilly Media, Inc., 448 pages, 2009.

Wang, C.; Said, M.; Gupta, A. Coverage Guided Systematic Concurrency Testing. In: *33rd International Conference on Software Engineering (ICSE)*, pages 221–230, 2011.

Wawersich, C.; Stilkerich, M.; Schröder-Preikschat, W. An OSEK/VDX-based Multi-JVM for Automotive Appliances. In: *Embedded System Design: Topics, Techniques and Trends*, volume 231 of *IFIP – The International Federation for Information Processing*, Springer, pages 85–96, 2007.

Weaver, V.; McKee, S. Can Hardware Performance Counters Be Trusted? In: *IEEE International Symposium on Workload Characterization (IISWC)*, pages 141–150, 2008.

Weyuker, E. J. On testing non-testable programs. *The Computer Journal*, volume 25, number 4, pages 465–470, 1982.

Weyuker, E. J. Data Flow Testing. In: *Encyclopedia of Software Engineering*, John Wiley & Sons, Inc., 2002.

Wilhelm, R.; Seidl, H. *Compiler Design: Virtual Machines.* Springer, 200 pages, 2011.

Wimmer, C.; Haupt, M.; Van De Vanter, M. L.; Jordan, M.; Daynès, L.; Simon, D. Maxine: An Approachable Virtual Machine for, and in, Java. *ACM Transactions on Architecture and Code Optimization (TACO)*, volume 9, number 4, pages 30:1–30:24, 2013.

Wimmer, C.; Haupt, M.; Vanter, M. L. V. D.; Jordan, M.; Daynès, L.; Simon, D. *Maxine: An Approachable Virtual Machine For, and In, Java.* Technical report 2012-0098, Oracle Labs, 2012.

Wirth, N. Recollections About the Development of Pascal. In: *History of programming languages – II*, ACM, pages 97–120, 1996.

Wohlin, C.; Runeson, P.; Höst, M.; Ohlsoon, M. C.; Regnell, B.; Wesslén, A. *Experimentation in software engineering: An introduction.* Kluwer Academic Publishers, 204 pages, 1999.

Wolfe, A. Toolkit: Java is Jumpin'. *Queue*, volume 1, number 10, pages 16–19, 2004.

Wong, W. E. *On Mutation and Data Flow.* Ph.D. Thesis, Purdue University, 1993.

WONG, W. E.; MALDONADO, J. C.; DELAMARO, M. E.; MATHUR, A. P. Constrained Mutation in C Programs. In: *Proceedings of the 8th Brazilian Simposium on Software Engineering*, pages 439–452, 1994.

WONG, W. E.; MATHUR, A. P. Reducing the Cost of Mutation Testing: An Empirical Study. *Journal of Systems and Software*, volume 31, number 3, pages 185–196, 1995.

WOODWARD, M. Mutation Testing – An Evolving Technique. In: *IEE Colloquium on Software Testing for Critical Systems*, pages 1–6, 1990.

WOODWARD, M. Mutation Testing – Its Origin and Evolution. *Information and Software Technology*, volume 35, number 3, pages 163–169, 1993.

WOODWARD, M.; HALEWOOD, K. From Weak to Strong, Dead or Alive? An Analysis of Some Mutation Testing Issues. In: *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 152–158, 1988.

WÜRTHINGER, T.; VANTER, M. L. V. D.; SIMON, D. *Perspectives of Systems Informatics*, chapter: Multi-level Virtual Machine Debugging Using the Java Platform Debugger Architecture, Springer Berlin/Heidelberg, pages 401–412, 2010.

XU, R.; WUNSCH, D. Survey of Clustering Algorithms. *IEEE Transactions on Neural Networks*, volume 16, number 3, pages 645–678, 2005.

ZHANG, L.; HOU, S. S.; HU, J. J.; XIE, T.; MEI, H. Is Operator-based Mutant Selection Superior to Random Mutant Selection? In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE)*, ACM, pages 435–444, 2010.