NOVA SCHOOL OF SCIENCE & TECHNOLOGY

DEPARTMENT OF COMPUTER SCIENCE

FILIPE ROBALO DE LUNA

Bachelor in Computer Science

OSCAR

A NOISE INJECTION FRAMEWORK FOR TESTING CONCURRENT SOFTWARE

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

OSCAR

A NOISE INJECTION FRAMEWORK FOR TESTING CONCURRENT SOFTWARE

FILIPE ROBALO DE LUNA

Bachelor in Computer Science

Adviser: João M. S. Lourenço
FCT — NOVA University Lisbon

Co-adviser: Jeremy Bradbury
Ontario Tech University

MASTER IN COMPUTER SCIENCE

NOVA University Lisbon
September, 2022

OSCAR

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v6.10.5) [133].

Acknowledgements

I wish to begin these acknowledgements by expressing my deepest thanks to both of my advisors, Professor João Lourenço and Professor Jeremy Bradbury. During the entire period of this dissertation's development, both of them went above and beyond the call of duty, providing the utmost dedication and support which were essential to this dissertation's success. I am deeply grateful for the opportunity to work with both Professor João Lourenço and Professor Jeremy Bradbury.

I also wish to express a special thanks to Professor Luis Russo, from IST (Instituto Superior Técnico) of Universidade de Lisboa and Professor Cláudia Soares, from FCT NOVA, for taking the time to provide me with valuable insight which helped me further the quality of this work.

Thanks should also go to Professor António Ravara, from FCT NOVA, for assisting me in expressing the work I have done throughout this dissertation in a more coherent manner, something which was more challenging than I would have expected.

I would like to extend my sincere thanks to FCT NOVA and all of its faculty for providing me with the highest standard of education throughout my entire tenure. Additionally, I would like to thank NOVA LINCS and the HiPSTr[1] project for all of the support, including the scholarship which was given to me in the context of this dissertation.

[1] HiPSTr, ref. PTDC/CCI-COM/32456/2017 e LISBOA-01-0145-FEDER-032456.

Abstract

"Moore's Law" is a well-known observable phenomenon in computer science that describes a visible yearly pattern in processor's die increase. Even though it has held true for the last 57 years, thermal limitations on how much a processor's core frequencies can be increased, have led to physical limitations to their performance scaling. The industry has since then shifted towards multicore architectures, which offer much better and scalable performance, while in turn forcing programmers to adopt the concurrent programming paradigm when designing new software, if they wish to make use of this added performance. The use of this paradigm comes with the unfortunate downside of the sudden appearance of a plethora of additional errors in their programs, stemming directly from their (poor) use of concurrency techniques.

Furthermore, these concurrent programs themselves are notoriously hard to design and to verify their correctness, with researchers continuously developing new, more effective and efficient methods of doing so. Noise injection, the theme of this dissertation, is one such method. It relies on the "probe effect" — the observable shift in the behaviour of concurrent programs upon the introduction of noise into their routines. The abandonment of ConTest, a popular proprietary and closed-source noise injection framework, for testing concurrent software written using the Java programming language, has left a void in the availability of noise injection frameworks for this programming language.

To mitigate this void, this dissertation proposes OSCAR — a novel open-source noise injection framework for the Java programming language, relying on static bytecode instrumentation for injecting noise. OSCAR will provide a free and well-documented noise injection tool for research, pedagogical and industry usage. Additionally, we propose a novel taxonomy for categorizing new and existing noise injection heuristics, together with a new method for generating and analysing concurrent software traces, based on string comparison metrics.

After noising programs from the IBM Concurrent Benchmark with different heuristics, we observed that OSCAR is highly effective in increasing the coverage of the interleaving space, and that the different heuristics provide diverse trade-offs on the cost and benefit (time/coverage) of the noise injection process.

Keywords: Noise Injection, Java, Bytecode, Instrumentation, Concurrency, Error Detection, Static Analysis

Resumo

A "Lei de Moore" é um fenómeno, bem conhecido na área das ciências da computação, que descreve um padrão evidente no aumento anual da densidade de transístores num processador. Mesmo mantendo-se válido nos últimos 57 anos, o aumento do desempenho dos processadores continua garrotado pelas limitações térmicas inerentes à subida da sua frequência de funcionamento. Desde então, a indústria transitou para arquiteturas multinúcleo, com significativamente melhor e mais escalável desempenho, mas obrigando os programadores a adotar o paradigma de programação concorrente ao desenhar os seus novos programas, para poderem aproveitar o desempenho adicional que advém do seu uso. O uso deste paradigma, no entanto, traz consigo, por consequência, a introdução de uma panóplia de novos erros nos programas, decorrentes diretamente da utilização (inadequada) de técnicas de programação concorrente.

Adicionalmente, estes programas concorrentes são conhecidos por serem consideravelmente mais difíceis de desenhar e de validar, quanto ao seu correto funcionamento, incentivando investigadores ao desenvolvimento de novos métodos mais eficientes e eficazes de o fazerem. A injeção de ruído, o tema principal desta dissertação, é um destes métodos. Esta baseia-se no "efeito sonda" (do inglês "probe effect") — caracterizado por uma mudança de comportamento observável em programas concorrentes, ao terem ruído introduzido nas suas rotinas. Com o abandono do ConTest, uma framework popular, proprietária e de código fechado, de análise dinâmica de programas concorrentes através de injecção de rúido, escritos com recurso à linguagem de programação Java, viu-se surgir um vazio na oferta de framework de injeção de ruído, para esta mesma linguagem.

Para mitigar este vazio, esta dissertação propõe o OSCAR — uma nova framework de injeção de ruído, de código-aberto, para a linguagem de programação Java, que utiliza manipulação estática de bytecode para realizar a introdução de ruído. O OSCAR pretende oferecer uma ferramenta livre e bem documentada de injecção de ruído para fins de investigação, pedagógicos ou até para a indústria. Adicionalmente, a dissertação propõe uma nova taxonomia para categorizar os diferentes tipos de heurísticas de injecção de ruídos novos e existentes, juntamente com um método para gerar e analisar traces de programas concorrentes, com base em métricas de comparação de strings.

Após inserir ruído em programas do IBM Concurrent Benchmark, com diversas heurísticas, observámos que o OSCAR consegue aumentar significativamente a dimensão da cobertura do

espaço de estados de programas concorrentes. Adicionalmente, verificou-se que diferentes heurísticas produzem um leque variado de prós e contras, especialmente em termos de eficácia versus eficiência.

Palavras-chave: Injecção de ruído, Java, Bytecode, Instrumentação, Concorrência, Deteção de erros, Análise estática

Contents

List of Figures

List of Algorithms

List of Listings

Acronyms

ACID Atomicity, Consistency, Isolation and Durability *(pp. 9, 10)*
API application programming interface *(pp. 6, 37, 38, 40, 63, 65, 70, 75, 108)*
AST Abstract Syntax Tree *(pp. 65, 70)*

CFLASH Concurrency Faults Localized Automatically using Search Heuristics *(pp. 34, 41)*
CLI command line interface *(pp. xiv, 75, 76, 78)*
CPU central processing unit *(p. 9)*

DCG directed cyclic graph *(pp. 49, 51–54, 72)*
DSM distributed shared-memory *(pp. 6, 7)*

ELF Executable and Linkable Format *(p. 36)*
ETH Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology) *(p. 22)*
ETRI Electronics and Telecommunications Research Institute *(p. 40)*

FERRARI Framework for Efficient Rewriting and Reification by Advanced Runtime Instrumentation *(p. 37)*

GNU GPL GNU General Public License *(p. 35)*
GUI graphical user interface *(p. 75)*

I/O input/output *(pp. 26, 73)*
IDE integrated development environments *(p. 18)*
ILP instruction-level parallelism *(p. 4)*
IPC inter-process communication *(pp. 6, 33)*

JDK Java Development Kit *(pp. 37, 47)*

jtp jimple transformation phase *(pp. 64, 65, 70, 71)*
JVM Java Virtual Machine *(pp. 37, 44, 47, 55, 56, 62, 68, 70, 109, 143)*

MPI message-passing interface *(pp. 29, 42, 110)*

NASA National Aeronautics and Space Administration *(p. 22)*
Ninja Network noise INJection Agent *(pp. 42, 110)*
NTP Network Time Protocol *(p. 8)*

OSCAR Open-Source Static Concurrency AnalyseR *(pp. 2, 3, 61, 62, 64–67, 69–76, 78, 79, 81, 83–89, 104–113, 133, 139–150, 152)*

P2P peer-to-peer *(p. 8)*
PEBIL PMaC's Efficient Binary Instrumentation Toolkit for Linux *(pp. 35, 36)*
POSIX Portable Operating System Interface *(pp. 27, 28, 36, 39)*
PRNG pseudorandom number generator *(p. 136)*
PROMELA Protocol Meta Language *(p. 23)*

SRR subgroup reproducible replay *(p. 110)*
SUT subject under test *(pp. 77, 87, 108, 112)*

TNCS test and noise configuration search *(pp. 32, 41, 88)*
TXL Turing eXtender Language *(pp. xiv, 34, 35, 41)*

UUID Universal Unique Identifier *(pp. 67–69, 73)*

VC verification condition *(p. 20)*

wjtp whole-jimple transformation phase *(pp. 64, 65, 70)*

Chapter 1

Introduction

1.1 Context

In 1965, Gordon E. Moore predicted that the transistor density of processor dies would double each year [148] and, 57 years later, this conjecture — more commonly known as *Moore's Law* —, still holds true. Unfortunately, due to thermal limitations, the processor clock frequencies saw their respective yearly growth diminish, reaching a 5 GHz ceiling almost a decade ago. This physical limitation to the processor's performance scaling made the industry shift towards multicore architectures as a means to scale the processing power of a processor die [102, 144]. Today, multicore processors are the *de facto* standard and dominate the server, workstation, personal and mobile processor market, with a significant amount being found in IoT systems as well. As multicore processors become ubiquitous, software must to be designed and/or optimized for these architectures, in order to make use of the potential performance gains stemming from the use of concurrency, as multicore processors execute computational tasks more efficiently by exploiting parallelism [82].

Historically, programmers and scientists elected the serial programming abstraction, mainly due to its simplicity [144]. This choice resulted in a plethora of tools, developed and perfected throughout the decades for sequential software, becoming insufficient. The industry has since then largely adapted itself, shifting towards the concurrent programming model and the use of multicore architectures, with these now being a standard feature present in all widely used programming languages [64]. However, the added complexity still comes at a great cost, as correct concurrent programs are notoriously harder to design and verify their correctness [5, 68, 163, 170, 173], ultimately resulting in additional errors [128].

1.2 Problem

Concurrent programs come with added performance and efficiency, through the coordinated usage of additional threads to execute independent sub-tasks, at the cost of added complexity and an overall increase in errors [202]. This stems from the fact that concurrent software is naturally non-deterministic, and it becomes impossible to predict the execution order of every

statement in every process or thread — their interleaving. In a regular sequential program, a developer only needs to concern himself with verifying safety and progress properties for a single sequence of instructions. Basically, it is possible to describe a sequential program as if having just a single "interleaving". Unlike sequential programs, which only depend on their input, concurrent programs depend on temporal events. As the number of possible interleavings for a concurrent program rises exponentially relative to the number of threads and instructions [58], standard testing techniques prove themselves ineffective against the massive number of possible interleavings, as some of these interleavings may have a very low probability of occurrence, being highly dependent of the (non-deterministic) thread scheduler [5, 163]. It is even possible for the same interleaving to occur repeatedly, giving the tester an illusion of correctness, as other erroneous interleavings, which may violate state invariants, remain untested.

1.3 Approach and Contributions

One approach to designing testing tools able to reliably tackle such massive amounts of interleavings, especially the ones with a low probability of manifesting themselves, is relying on a witnessable behaviour of concurrent software known as the *probe effect* [71]. Essentially, the *probe effect* is the observable difference in a concurrent program's behaviour when behaviour perturbation is introduced. The *probe effect* can be leveraged to our advantage, by introducing delays in specific sections of a program. This, in turn, influences the scheduler, resulting in different interleavings that, although possible, most probably would not be witnessed otherwise [63]. One such application of this technique is "noise injection", which relies on the insertion of additional code into a program, that introduces said delays, either randomly or based on a heuristic, into a program's routine [116]. This additional code can be introduced statically, resorting to instrumentation tools or dynamically, at run time.

This dissertation makes an in-depth study of the state of the art in the field of noise injection and proposes novel heuristics and techniques for trace generation and analysis. Built from these insights, come the two major contributions of this dissertation — the design, implementation and evaluation of an all-new shared-memory-oriented noise injection framework for concurrent software written using the Java programming language; and a method for measuring differences between interleavings based on classical string comparison algorithms.

This tool, designated as Open-Source Static Concurrency AnalyseR (OSCAR), will inject noise by means of intermediate code instrumentation into Java bytecode files, with a scalable architecture allowing for a seamless addition of support for more "noisable" Java concurrency primitives with different noise injection heuristics and trace recording for posterior analysis. Being fully open-source, OSCAR will fill an existing void in the availability of noise injection frameworks for concurrent software written using the Java programming language.

1.4 Document Structure

The remainder of this dissertation is structured as follows:

Chapter 2: This chapter is responsible for laying the foundations for this dissertation, presenting a more in-depth introduction to concurrent programming, including its perks and caveats, its particular bug types and how to detect them, with a few real-world examples of notable technologies and techniques.

Chapter 3: In this chapter, it is possible to find a more in-depth study of noise injection techniques and the progress in this field made by the scientific community and a presentation of existing practical implementations of this technique. This chapters fills the purpose of gathering insight into the current state of the art, which may then be used in our own solution.

Chapter 4: This chapter applies all of the lessons learned from the previous chapter, which went over the state of the art in the field of noise injection, to describe how to apply noise injection specifically to Java software. During this chapter, novel heuristics and trace generation and analysis approaches are presented.

Chapter 5: This section will describe the noise injection framework tool which was developed to incorporate all the insights from Chapters 3 and 4 — OSCAR. The chapter goes through the architecture, interfaces and specificities of how the techniques developed in Chapter 4 were implemented.

Chapter 6: This chapter contains the validation of OSCAR, both functional and non-functional, as well as the evaluation of the OSCAR noise injection framework.

Chapter 7: This chapter contains the conclusion, stemming from all the insights which were obtained throughout the development of this dissertation. Additionally, the second part of the chapter proposes future work for the OSCAR noise injection framework.

Chapter 2

Background

2.1 Concurrent Software

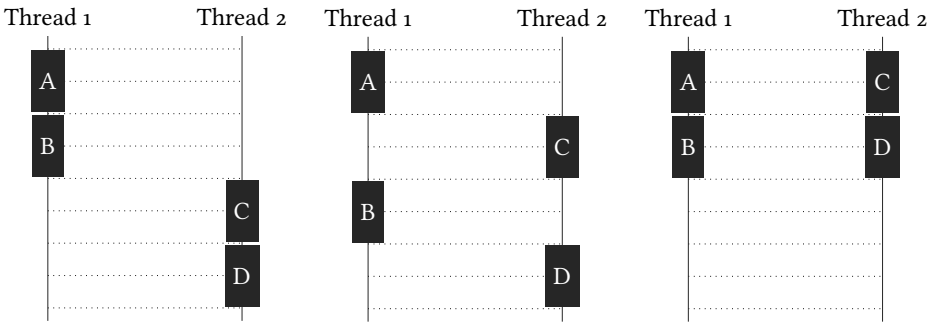
A sequential program can be expressed as a single sequential state machine — a process executing a single control flow. A concurrent program differs from a sequential program, by having multiple processes or threads, each sequentially executing its own state machine, cooperating towards an end goal [2] through the use of some type of communication (e.g., exchanging messages) [164]. This form of interaction between processes is called synchronization and will be further discussed, along with the different forms of achieving it, later in Section 2.1.2.

In most operating systems, a process is an abstraction technique for isolating executing programs relying on hardware mechanisms. The operating system has the responsibility of allocating the process the set of its needed resources and ensuring these are isolated from other processes [1]. A thread, however, is an abstraction inside a process, which shares its memory and resources [123] with other threads in this same process.

The concept of parallelism is very similar to that of concurrency, which contributes to them being, albeit erroneously, commonly used interchangeably. Parallelism differs from concurrency, with a system being said to be parallel if it can run two actions simultaneously and independently [23], whereas a concurrent system may depend on some sort of ordering or external synchronization between different actions. Another difference stems from the fact that a program running on a single core can contain concurrency, by leveraging thread instruction ordering, but a parallel program requires either multiple cores, processors or some other medium through which tasks can be executed simultaneously, such as a processor with support for instruction-level parallelism (ILP).

Figure 2.1 illustrates the contrast between the different ways to execute a process running two threads. Each of its threads is represented by a different colour — either black or white, and contains an ordered set of actions or statements. In Figure 2.1(a), it is possible to observe two threads, each running its own set of actions sequentially, one at a time. Figure 2.1(b) shows two concurrent threads alternating between each other during execution. Lastly, Figure 2.1(c) illustrates a parallel execution, where both threads are running simultaneously and independently.

The parallelization in this particular example is particularly advantageous, as the throughput is directly proportional to the thread count.



(a) Sequential Execution (b) Concurrent Execution (c) Parallel Execution

Figure 2.1: Visualizing sequential, concurrent and parallel execution.

As mentioned earlier in Section 1.2, concurrent programs are non-deterministic by nature, meaning it is impossible to predict the execution order of their actions. Each single one of these unique executions is called an interleaving [23]. While a sequential program's result depends uniquely on its input, a concurrent program's result will also depend on temporal events. Together, the input and temporal events will make the execution order of the program result on a specific interleaving. Figures 2.1(a) and 2.1(c), are actually valid interleavings for a concurrent program, with the last one requiring at least two processors (for simultaneous execution). The justification for this last phenomenon is the fact that threads are asynchronous and have no timing guarantees [82], meaning all the executions illustrated can have an infinite number of different timings. This further demonstrates the non-determinism that characterizes concurrent execution and how challenging it becomes to build correct solutions to deal with its issues [101].

In other words, in concurrent programs, the number of possible interleavings grows at an exponential rate, relative to the number of threads and instructions of the program — a famous problem known as the *state space explosion*. Code coverage, a function of how many of the program's possible states are covered in one or more given runs, tends to also decrease exponentially.

In the event that a system has more threads than processors, concurrency becomes possible by resorting to *time slicing* — a technique in which a processor splits the execution of two or more threads between a single processor.

2.1.1 Concurrent Software Paradigms

When designing concurrent software, programmers mostly resort to one of two main concurrent programming paradigms [51] — shared-memory and message-passing. During this section, both of these models will be approached. These define the synchronization medium, only becoming relevant in systems with more than one thread running simultaneously. This is due to the fact that a single thread program will never have simultaneous accesses to a shared resource, rendering synchronization unnecessary, as can be further proven by resorting to Lamport's *happens-before* relation [121]:

1. The notation $a \rightarrow b$, means an event a happened before another event b, and every thread in a system agrees to this ordering [139];
2. This relationship is transitive, meaning $a \rightarrow b$ and $b \rightarrow c$ imply $a \rightarrow c$;
3. Any two events are said to be concurrent if $a \nrightarrow b \wedge b \nrightarrow a$;
4. But, since only one thread exists: $\forall\ a,\ b : a \rightarrow b \vee b \rightarrow a$, as property 1 guarantees that a single thread would always agree with its own ordering, as a total order exists in the entire set of events.

Figure 2.2: Proof using the happens-before relation.

2.1.1.1 Shared-Memory

The shared-memory model is defined by the medium through which synchronism between different threads is achieved. In a shared-memory program, multiple asynchronous threads communicate by calling read and write methods over registers that reside in a shared memory addressing space, from which both can read and write to [23, 82]. It is very straightforward to implement a shared-memory model when using threads, since threads all share part of the memory in a process. The challenges with shared-memory synchronization implementations occur when multiple threads try to access the same memory location at simultaneously.

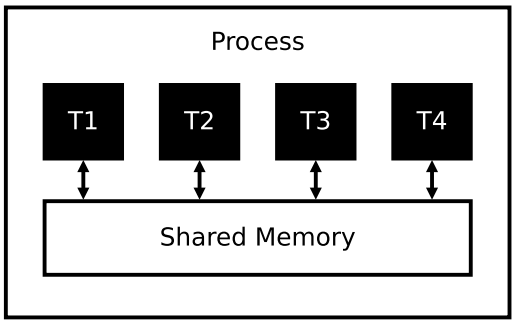


Figure 2.3: A process with 4 threads in a shared-memory configuration.

However, it is possible to achieve a shared-memory configuration between different processes as well, by resorting to abstractions that simulate a shared-memory addressing space. For instance, different processes may access a common file through the use of the inter-process communication (IPC) channels of an operating system. Another example is distributed shared-memory (DSM), where different machines can access a shared address space through a regular load/store application programming interface (API), by making use of a distributed system that interconnects all of them [205].

2.1.1.2 Message-Passing

Message-Passing is the model of choice for building modern large-scale high-performance computing applications, such as supercomputers and data centres [172]. In the message-passing model, synchronicity between different processes is achieved by a shared bus through which messages are exchanged. Processes wanting to communicate with each other, do so through the

use of channels that allow each process to send and receive messages [2]. The most notable way in which this bus can be implemented, is through a network, allowing a system to become easily scalable. Message-passing systems are more often called distributed systems [9, 165], as they effectively distribute their resources and data between them in order to cooperate towards a common goal. Distributed systems rely on distributed algorithms, which are extremely complex to design and implement, earning them their own field in computer science — Distributed Systems. These distributed systems often rely on asynchronous communication channels, which introduce non-determinism in communication patterns, increasing the implementation and debugging difficulty [172]. This non-determinism may introduce bugs specific to the message-passing paradigm, such as message races.

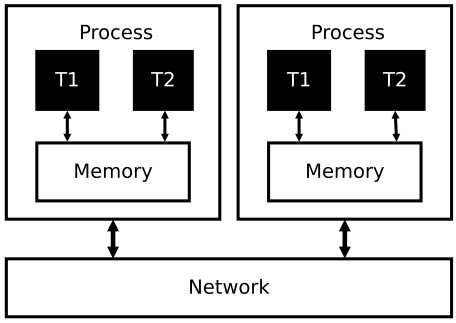


Figure 2.4: Two processes with 2 threads each, in a message-passing configuration.

Although generally being considered more error-prone than the shared-memory model [139], some use cases require or greatly-benefit immensely from a message-passing model. Such an example is a user-oriented service in which private variables cannot be shared [2]. Another example are systems that need to be easily scalable, such as peer-to-peer networks (e.g., *Chord* [182], *Kademlia* [143]) and blockchains (e.g., *Bitcoin* [150], *Ethereum* [32]). Furthermore, message-passing systems can be used to implement a DSM algorithm on top.

The *CAP* theorem [25], is a conjecture created by Brewer, in which he states a distributed system can only have, at most, two of the following three properties [24]:

1. **Consistency**: the system behaves as if having but a single copy of all data, serving up-to-date copies of all data at every node;
2. **Availability**: the system behaves correctly, even in the presence of node failures;
3. **Partition-Tolerance**: the system can withstand network partitioning, which occurs when network failures cause node isolation.

As such, a programmer needs to take this conjecture into account, when designing a distributed algorithm, leveraging between the pros and cons of each property as suitable for its use-case. *CAP* plays an important role in replication, one of the main areas of interest in distributed systems. Replication has two main purposes: performance and fault-tolerance [195].

Message-Passing systems can either be qualified as synchronous or asynchronous, depending on the techniques they use in order to achieve synchronism.

A distributed system is said to be synchronous when it employs a global clock to ensure an order of events, through some form of fault-tolerant synchronization algorithm [111, 112]. This form of global clock can either be tied to real-world physical clock or a logical clock implementation [139].

Examples of distributed algorithms using physical clocks include: the Network Time Protocol (NTP) [44, 147], which employs a centralized clock in the form of a time server which other processes can query in order to know the time; and the Berkeley algorithm, which assumes no process has an accurate clock and instead, using a time server to constantly poll every process, compute an average time and tell each process to adjust their clocks accordingly [75].

Likewise, examples of distributed algorithms using logical clocks include: *Lamport Timestamps* [121], based on the happens-before algorithm mentioned in Figure 2.2, enforces a partial ordering through independent counters at each machine, whose values are timestamped on messages exchanges for synchronizing; and the *Vector clock* algorithm, first mentioned by Liskov [132], which is a spiritual successor to the Lamport Timestamps in which each process keeps every process' (including his) last received counter value and updates it through the exchange of multipart timestamps, allowing the system to detect causality between events [13].

In asynchronous distributed systems, processes have independent clocks and thus different views of the global state, transitioning merely in response to inputs [45]. There are also no upper-bounds on how long a message can take to be delivered and the number of steps until state convergence can be achieved, which can lead to stale values. *Gossip* protocols, which are heavily used in peer-to-peer (P2P) communication, such as *HyParView* [125], are an example of asynchronous distributed algorithms which allow for decentralized scalable communication.

2.1.2 Synchronization

Synchronization describes a plethora of different algorithm design techniques that can be employed in a concurrent program, as a means of preventing incorrect interleavings [2, 174]. Any synchronized interaction between two processes falls into one of two categories — competitive or cooperative [164]. In competitive synchronization, two or more processes "race" in order to obtain control of a shared resource they cannot access simultaneously, with the algorithm being responsible for enforcing this mutual exclusiveness. However, in cooperative synchronization, some processes must wait for other processes' tasks to complete, before continuing execution.

The correctness of a synchronization algorithm must guarantee two properties — safety and liveness [174]. A common simplified explanation for these two concepts is the following mnemonic: "Safety asserts that bad things never happen, while liveness asserts that something good eventually happens" [119]. In safety, the "bad thing" refers to the program violating an invariant and reaching a bad state. For liveness, the "good thing" refers to the program eventually finishing — never reaching a state where it stops or loops indefinitely. All concurrency-related errors stem from invalid synchronization implementations.

Many synchronization primitives rely on the atomicity property, which guarantees that there cannot be any malign repercussion from two or more threads simultaneously accessing a shared resource or invoking an operation [69]. In other words, atomicity violations occur when there is an erroneous assumption about the atomicity scope of a code block, incorrectly splitting it into two or more other atomic blocks, thus allowing unintended interleavings [54].

It is possible to consider the notion of atomicity in concurrent software equivalent to the *Atomicity* and *Isolation* in the Atomicity, Consistency, Isolation and Durability (ACID) properties of transactions, which will be presented in Section 2.1.2.2, together with the *Ordering* property, which was described through the *happens-before* algorithm in Figure 2.2. Breaking a single one of these properties will violate the program's atomicity in the respective blocks.

Atomicity in concurrent software can be achieved either by the means of hardware support, such as a processor instruction that can read and write simultaneously, or by resorting to a software solution such as Lamport's *Bakery Algorithm* [120]. Over the next few pages, a few examples of common and widely-used synchronization primitives will be presented.

2.1.2.1 Hardware Primitives

Hardware primitives are part of a processor's instruction set and cannot be implemented at software level. These are essential, as they allow for fast atomic operations for higher level synchronization implementations. Several of these instructions exist and are included in most commercially available central processing units (CPUs) [164], a few notable examples are:

Test&Set: Sets a register to 0 and returns its previous value, or resets it to its initial value of 1;

Swap: Assigns a chosen value to a register and returns its previous value;

Compare&Swap: It receives two parameters, an old and new value, and if the register's value matches the old one, it is updated to the new value. Lastly, it returns a boolean value informing if the operation was successful;

Fetch&Add: This instruction receives two parameters: a value and an address. It increments the value stored at the given address, by the value given as a parameter, returning the original value afterwards.

2.1.2.2 Software Primitives

Semaphores

A semaphore [55] is one of the earliest synchronization methods, presented in 1965 by Dijkstra. Its basis is a shared variable (usually an integer) that represents the number of available resources. The initial value of a semaphore can be any positive integer, since a value of 0 would mean it has no available resources and any thread would loop indefinitely. Following its creation, a semaphore only has two possible interactions — increment and decrement the variable [10]. A simple implementation of a lock is a binary semaphore, where the variable can only have the values 0 or 1.

Upon reaching a shared resource, threads enter a loop in which they keep checking this variable, waiting for a resource to become available. When it does, the thread decrements the variable, signalling that a resource has been taken, and initiates the procedure. After completion, the same thread will increment the variable, freeing a resource.

However, for semaphores to work, the increment and decrement operations must be atomic, lest multiple threads decrement the variable simultaneously and have it reach negative values.

Barriers

A barrier, also called a *rendeszvous* [164], is a synchronization method relying on a control point that halts all threads upon arrival. These same threads will then keep looping until every single one of them has arrived to that control point.

Its implementation is very straightforward, as all that is necessary is a shared atomic integer that is incremented by each thread upon arrival [2]. When the integer reaches n, with n being the number of threads, it will unlock and the threads will resume their routine.

Transactions

Transactions are a high-level form of synchronization, stemming from the fact that regular locks are not scalable or composable [164]. Essentially, two sequential atomic operations have a non-atomic interval between them, which can cause a myriad of synchronization issues.

A transaction is a software abstraction that allows a thread to execute a collection of operations while ensuring the ACID properties [98]. Common transaction implementations start encompass the sequence of operations between two keywords — begin and end. The meaning of each letter of the ACID acronym, is the following:

Atomicity: any and all of the transaction's changes to a program's state are atomic, meaning they either all happen or none happen;

Consistency: the transaction's routine itself is correct and cannot violate any of the state's integrity constraints;

Isolation: any concurrent transactions must appear to each other as if executed either before or after the other;

Durability: any changes caused by a successful transaction must persist, even through failures.

All these properties were described for the simplest transaction system, where each thread controls the entire state during a transaction. However, modern transaction systems, found on modern database systems, first discriminate the affected state locations and guarantee the ACID properties only for those specific locations, allowing simultaneous processing of transactions.

Condition variables

Condition variables, despite being called "variables", are effectively queues [164, 168]. These allow a thread to wait until a resource with a lock currently held by another thread, meets a condition [144]. They are implemented as objects threads can interact with, through the *wait* and *signal* operations. *Wait* is self-explanatory — a thread invokes *wait* when the condition is not as desired, stops executing and waits inside the queue. *Signal* is used by a thread to notify others of a change program that may affect the condition. It dequeues one of the waiting threads, lets it proceed, and waits for it to complete its routine, so it may reacquire the lock [164].

Monitors

The monitor is another high-level primitive, invented by Brinch and Hansen [26] and further developed by Hoare [85]. A monitor is a concurrent object representing a set of resources, methods to access said resources and also its own scheduler, combining synchronization and data in a single package [2, 82].

Monitors make use of condition variables [2] to ensure mutual exclusion when accessing any of its methods [164]. Upon trying to acquire a lock, a thread can choose to either spin or block. These last two are fundamental concepts of concurrent software programming.

Spinning, also known as busy-waiting [174], is a technique that makes a thread loop, while constantly checking for possession of a lock, so it can then acquire it and proceed. Spinning is useful, as the thread will be ready to execute the task as soon as it is free. The downside being that, while the thread is spinning, it is consuming a physical processor and resources that could otherwise be used by another thread.

On the other hand, blocking makes a thread sleep and wait for a signal to wake and proceed. This allows another thread to use its resources while it waits, with the time-consuming penalty the stopping and restarting procedures, also known as the context-switch overhead [49, 105].

Ideally, one should opt to spin a thread when it is known that there will be a short waiting time and block when the waiting time is expected to be long [82, 174].

2.1.3 Concurrency Errors

Designing concurrent software comes with added complexity and an increase of errors, with many programmers considering them to be amongst the most insidious [101]. But these errors only become an issue when they violate the invariants a programmer defined for its program. A program can be perfectly correct in the presence of concurrency bugs. Polling implementations, due to their continuously overwriting nature, are an example of a solution that can fall into this category.

While all concurrency errors stem from invalid or non-existent synchronization implementations, as stated earlier in Section 2.1.2, these can be broken down relatively to their specific causes and effects — a *taxonomy* —, with many authors adapting different models. During research for this dissertation, the following taxonomy was developed for describing the various concurrency bugs.

2.1.3.1 Low-Level Data Races

Data races can be categorized into two types — low-level and high-level [5]. The former, low-level data races, will be the first to be described, with the latter being presented in the following section.

When designing concurrent software, low-level data races are amongst the most common sources of errors a programmer will come across [5, 114]. Low-level data races occur when two unsynchronized threads attempt to simultaneously access a shared resource, with at least one of these accesses involving a write operation [5, 23, 54, 128, 173]. In other words, they occur when the atomicity property is violated, which may result on our program becoming non-deterministic, as different interleavings may yield different results. Bernstein presented a method [16], consisting of a set of conditions, for verifying if two successive sections of code can be parallelized (i.e., have no dependencies).

If one were to assume two statements/instructions, represented by S_n, and two variables, represented by R_n and W_n, which are the sets of variables or memory locations these statements/instructions read and write to, respectively, such that:

$$S_1 \longrightarrow S_2, n \in \{1, 2\}$$

Then, a statement S_2 depends on S_1 when any of these conditions, except the last, are met:

- $W_1 \cap R_2 \neq \varnothing$: true/flow dependency, where S_2 reads a value written by S_1;
- $R_1 \cap W_2 \neq \varnothing$: anti-dependency, where S_2 overwrites a value read by S_1, thus making S_1's value stale;
- $W_1 \cap W_2 \neq \varnothing$: output dependency, where both S_2 and S_1 write to the same location, leading to an inconsistent state;
- $R_1 \cap R_2 \neq \varnothing$: subsequent reads do not cause a dependency, thus making them thread-safe (i.e., concurrent calls do not cause any side-effects) [23].

In short, code that contains unsynchronized dependencies, as shown in Listing 2.1, will contain at least one interleaving which will lead to a data race when executed concurrently. In other words, the atomicity property needs to be guaranteed, when accessing a variable, shared resource or memory location.

```
a = 1;
b = a;
```

(a) True Dependency

```
a = b;
b = c;
```

(b) Anti-Dependency

```
a = 1;
a = 2;
```

(c) Output Dependency

```
a = c;
b = c;
```

(d) No Dependency

Listing 2.1: Data dependencies examples.

```
void react_to_post(Post &post, int vote) {
    if (vote == 1)
        post.score++;
    else
        post.score--;
}
```

Listing 2.2: Data race example: Voting for a forum post.

Listing 2.2 shows a real world example of how data races can affect software. Modern online forums are platforms which allow users to submit content and vote positively or negatively in other users' content, in order to reflect the quality and/or relevancy of posts. For this example, the assumption exists that this code is being run by multiple threads simultaneously. If two threads attempt to update the `post.score` variable simultaneously, there will be multiple possible outcomes depending on the interleaving. This stems from increment operations not guaranteeing atomicity, due to being composed by two instructions — a read followed by a write. As the processor scheduler is non-deterministic, interleavings will exist where, between one thread's read and write instructions, another thread reads and updates the variable. In a scenario with 1000 threads simultaneously executing this operation, the `post.score` variable value could be incremented between 1 and 1000 times. Due to the number of interleavings exponentially increasing as a program grows, data races can be incredibly hard to detect, as some of these interleavings may need very special conditions in order to manifest themselves, thus, in some cases, rendering regular testing techniques, like unit testing, futile [5].

2.1.3.2 High-Level Data Races

High-level data races, also called atomicity violations [54], occur when a sequence of operations that access shared data are inserted in a mutual-exclusion block (e.g., protected by a lock), but faulty program logic subjects them to interleavings with unintended results [6]. Since any single read or write operation is, by itself, an atomic block consisting of a single operation, these may also cause high-level data races when paired with another atomic block with multiple operations.

An example of a high-level data race can be a series of simultaneous atomic updates to multiple values in an object, but not all of them being in the same critical section, allowing for interleavings resulting in data corruption. Based on how they manifest themselves, it was chosen to break down high-level data races into three types: atomicity violations, order violations and stale value errors.

Atomicity Violations

After examining multiple concurrency bugs found in real world software, Lu et al. [134] verified that order and atomicity violations account for 97% of all concurrency bugs not related to deadlocks or 66% of all concurrency bugs. Both of them result from incorrect assumptions of how code is run concurrently, since they do not manifest themselves when the program is run with only a single thread.

```
Struct Menu { char dish[40]; };

void update_menu(Menu &new_menu) {
    for (int i = 1; i < 40; i ++) {
        pthread_mutex_lock(&mutex);
        menu[i] = new_menu[i];
        pthread_mutex_unlock(&mutex);
    }
}
```

Listing 2.3: Atomicity violation example: Updating the menu.

Atomicity violations are a type of concurrency bug which is notoriously challenging to both detect and fix [134, 136, 157], occurring when programmers fail to properly identify and isolate inside critical sections, atomic or non-atomic statements or blocks of code. In listing 2.3 one can observe a real world example of an atomicity violation bug. A low-level data race does not exist because all accesses to shared variables are correctly synchronized. Instead, the bug occurs because the programmer failed to properly encapsulate all the dish assignments to `menu[i]`, into the same atomic block. This incorrect synchronization implementation can ultimately result in corruption of the menu itself, since a parallel execution allows for interleavings where the entries on new menu are a mixture of all the menus currently being assigned simultaneously.

Order Violations

Order violation errors stem from the improper or inexistent ordering of statements or atomic regions of code by means of synchronization, which in turn will result in unexpected and interleavings, which themselves may be incorrect [134].

```
char data[MAX_DATA_SIZE];
pthread_t  fetcher_id;
pthread_t  parser_id;

pthread_create(&fetcher_id, NULL, fetch_data, (void *) data);
pthread_create(&parser_id, NULL, parse_data, (void *) data);
```

Listing 2.4: Order violation example: Parsing a website's data.

In listing 2.4, it is possible to observe another piece of mock code using one thread to fetch a website's data and another to parse it. The data itself is not important, it could be, for example, financial or meteorological. The real issue is that there is a clear order violation, because unwanted interleavings that parse the not yet initialized (*null*) data exist. As such, there will be a need to enforce some order of execution by means of a synchronization mechanism.

Stale Values

Lastly, stale value errors also occur in the presence of atomic memory accesses. These errors will manifest themselves when a thread holds a lock when accessing a shared variable, but not

afterwards, when effectively using its value in some operation [31]. This type of error can be especially disastrous if a value is, for instance, a pointer/reference to another value.

```
struct RegistedTemperature { int temp; int sensor; };

RegistedTemperature temperatures[MAX_TEMPS];
int temperature_count = 0;
int * current_temp = malloc(sizeof(int));

while (1) {
    spawn_threads(6); // Spawn 6 threads to concurrently execute this block

    // Each thread will access a different sensor
    int sensor_id = get_random_sensor_id();

    pthread_mutex_lock(&mutex);
    int read_temp = probe_temp_sensor();
    *current_temp = read_temp;
    pthread_mutex_unlock(&mutex);

    pthread_mutex_lock(&mutex);
    temperatures[temperature_count].temp = read_temp;
    temperatures[temperature_count].sensor = sensor_id;
    temperature_count++;
    pthread_mutex_unlock(&mutex);
}

printf("%d", &current_temp)
```

Listing 2.5: Stale value example: Temperature sensor array.

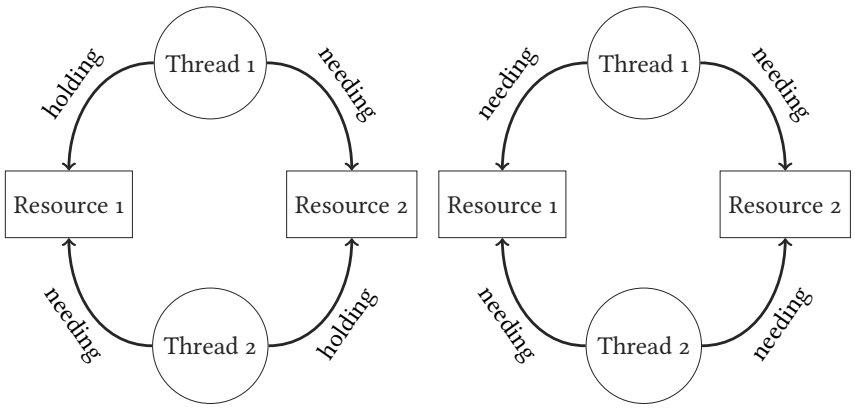
Listing 2.5 shows a mock program that concurrently queries random temperature sensors in an sensor array and registers their current reading. It is observable that the write access to the `current_temp` variable is atomic and so is its reading, when assigning its value to `read_temp`. But, after the first atomic block, the thread loses its lock on the access to the `current_temp` variable, meaning its value can be updated by another thread, since it is a pointer. This interleaving will result in the `read_temp` becoming out-of-sync with the `current_temp`, when being saved, leading to a stale value.

2.1.3.3 Deadlocks and Livelocks

Deadlocks are a phenomenon that occurs in concurrent algorithms that employ cooperative synchronization. They occur when at least two jobs cannot be completed due to conflicting resource requirements [177], thus forcing threads to remain blocked indefinitely, unable to proceed [88, 139]. Cooperative synchronization implementations, as mentioned in Section 2.1.2, make use of coordinated locks between processes to achieve mutual exclusion. A poor implementation will ultimately lead to situations in which the severity of the lack of coordination results in circular dependencies that are impossible to satisfy. Interestingly, a single thread can deadlock itself, by requesting a resource it already holds, a phenomenon that comprises almost 1/4 of all deadlock bugs [134] The four conditions a synchronization algorithm must meet, for a deadlock to take place[23, 177], are the following:

1. Any individual resource can be only held by one thread at a time;
2. Any thread already holding a resource may attempt to grab hold of another;
3. Any resource held by a thread must be voluntarily released by that same thread;
4. A circular dependency of resource requests between threads can occur.

Any program can become deadlock-free by ensuring at least one of the aforementioned conditions is not satisfied.



(a) Deadlock example (adapted from [106]). (b) Livelock example

Figure 2.5: Deadlock and Livelock examples.

An example of a deadlock situation can be seen in Figure 2.5(a). This example shows two threads, each one holding a resource the other thread wants while waiting for the other's resource to become free. As a thread will spin indefinitely until it can attain a resource it wants, thus these threads will never recover from this situation. This scenario will, most likely, ultimately result in a program failure.

A livelock, the deadlock counterpart, is the type of other error stemming from a poor cooperative synchronization implementation. Many definitions exist of what a livelock represents [83], but for simplicity, a more common notion will be adopted: a livelock is a situation in which threads are actively trying to progress but cannot due to a synchronization fault [23]. This highly contrasts deadlocks, where threads cannot even attempt to progress, since they are stuck in a wait state.

The illustration seen in Figure 2.5(b), demonstrates a typical livelock scenario. The two threads do not hold any resource at the moment and both need to exclusively attain both resources. When a thread attempts to obtain a resource and manages, the other will obtain the other and both will reach a state where they must free their resources because they cannot hold both simultaneously. This negotiation strategy, does guarantee deadlock-freedom but can still carry on forever, with both processes never progressing. In contrast, a program that will never run into a livelock situation is said to be *livelock-free*.

2.1.3.4 Starvation

A thread is said to be starving if, upon reaching a critical section, no resources are ever allocated to it due to being constantly bypassed by other threads [8], leaving it unable to progress indefinitely. To better understand starvation and what it entails, it is necessary to first delve into the concept of liveness. Liveness was a concept mentioned previously in Section 2.1.2, which described the ability of a concurrent program to consistently progress. In regard to their liveness properties, lock-based concurrent software falls into three categories [164]. Each of these categories provides stronger guarantees of progress over the latter's:

Deadlock-freedom This property ensures that at least one thread in the entire system will eventually acquire a lock and make progress, after reaching any critical section zone. It is required to ensure mutual exclusion [82].

Starvation-freedom Since deadlock-freedom will only guarantee progress for one thread, it cannot guarantee an upper bound to how many threads will be starving or deadlocked. Starvation-freedom (also called "wait freedom" [174] or "finite bypass" [164]), is a stronger property that assures all threads in the system will eventually progress (i.e., none will be starving). Still, it cannot guarantee a lower bound on the amount of time it will take for a thread to progress, meaning a thread may take an infinite amount of time until it progresses. Thus, achieving starvation-freedom may not be a very compelling endeavour [82], as the added overhead cannot always be justified.

Bounded bypass Instead of only guaranteeing some thread will eventually progress, bounded bypass guarantees an upper bound to the number of tries until a thread manages to progress from a critical section. Raynal [164] presents some synchronization algorithms, such as *Peterson's Algorithm*-[159], that implicitly ensure bounded bypasses, by relying on a mechanism that prevents the same processes from repeatedly acquiring locks.

2.2 Concurrent Software Testing

It has already been previously established, in this document, that opting to develop programs using the concurrent paradigm, will usually yield considerably better performance at the cost of the appearance of additional errors that are specific to this paradigm, stemming directly from the fact that concurrency is non-deterministic in nature. Tai [184] reasoned that unit-testing a concurrent program with a typical method of comparing an output, given the same input, is not a valid method, due to the following three reasons:

1. However many times you successfully execute a program with the same input, there is no guarantee it will not produce a different result in the future;
2. After correcting a previously detected error and successfully executing the program afterwards, multiple times, it is still possible that the error will manifest itself in a subsequent execution;

3. Likewise, after introducing a new feature and successfully executing the program afterwards, there is no guarantee that this feature will not trigger additional errors.

Even though it is already implied in these conditions, it is also important to stress that multiple repeated runs of a concurrent program can consistently result in the same interleaving. This conjecture is directly related to the state space explosion problem, which was first stated in Section 2.1. As the number of possible interleavings grows, relative to the number of instructions and threads running the program, it becomes increasingly difficult to guarantee coverage of the whole state space. From Tai's conjecture, it is possible to infer that different interleavings may have different chances of occurring. This also means that some interlavings may be extremely "rare", thus hiding errors from developers and testers, representing the main challenge to concurrent software testing. As such, concurrent software requires testing methods specifically tailored to tackle its non-deterministic nature.

This section will be divided into two subsections: the first subsection will describe how concurrency testing solutions are implemented, according to their different analysis types; the second and final subsection, will describe what methods and tools are used to detect the errors that were previously described in Section 2.1.3.

2.2.1 Software Analysis Types

This section will focus on describing the different approaches to how a piece of concurrent software can be analysed, for example, by analysing a program's source code or its output, given an input, in order to deduce errors. Furthermore, the section will look into what techniques, specific to concurrent software, can be used to implement said type of analysis, and the particular drawbacks that different types of analysis come with.

2.2.1.1 Static Analysis

Static analysis is a process that evaluates a given piece of software based on form, structure or documentation [95], instead of its behaviour [151]. Throughout this dissertation, static analysis tools shall be defined as those that infer errors through source, intermediate or machine code analysis, rather than by the analysis of its behaviour during its run time.

Many modern interpreters or compilers, for the most used programming languages nowadays (e.g., C, Java, Python and JavaScript), have some sort of built-in static error analysis by default [151]. Additionally, most modern integrated development environmentss (IDEs), also have their own static analysers to perform additional checks [162]. These tools usually resort to some sort of hierarchical tree-like representation of the program, as this technique often makes parsing more convenient [100]. These trees are then traversed recursively by the compiler or interpreter to allow for lexical, syntax and semantic code analysis. While analysing, several errors may be detected (i.e., compile time errors), like improper variable type casting, lack of braces or semicolons and out-of-scope variable access. Even though static analysis has become virtually invaluable for a modern developer, it is limited to the amount of rules and patterns it is programmed to find, and,

additionally, its predicting nature is known to result in a lot of false positives and negatives [35]. One of the main perks of static analysis, is completeness. Unfortunately, this comes at the cost of including unnecessary infeasible paths in its analysis [12].

Static analysis, usually suffers, by design, of being more limited in the size of the program scopes it can analyse effectively and efficiently, whereas dynamic analysis can examine an extremely long path, allowing it to find violations of invariants between dependencies that are "farther" apart in the program [12].

If one wishes to extend the existing static analysis with logic capable additional and more complex deductions of program errors, it is possible to either piggyback the existing compiler parsing step or, instead, create an entirely new tool, with its own parsing logic, that creates its own tree-like structure and infers new types of errors from it.

Unfortunately, static analysis becomes increasingly more complex when dealing with concurrent software [145, 173]. This is mainly due to concurrent software being inherently nondeterministic, resulting from the enormous amount of potential interleavings amongst its threads [103].

2.2.1.2 Dynamic Analysis

Not all errors can be deduced just from examining the source code and applying logical reasoning (e.g., *Hoare logic* [84]) to verify correctness, meaning a different approach to testing is necessary. Dynamic analysis, is, in its essence, the opposite of static analysis, as it evaluates a piece of software based on its behaviour during or after execution [12, 95, 151]. Dynamic analysis is vital, as some errors only manifest themselves during run time — these are also known as run time errors — , through very specific input (e.g., edge-cases) or execution paths, which allows us to detect new invariants that the static analysis could not. Thus, dynamic analysis techniques monitor a program's execution, reasoning about its behaviour and issuing warnings about detected and even possible errors [64], usually inferring this knowledge from the witnessing and subsequent processing of multiple runs [63].

While static analysis, as a technique, could boast completeness, dynamic analysis cannot. Since its observations are only valid for specific executions, dynamic analysis is always limited to the amount of program paths and inputs which are indeed tested [12]. This can lead to situations where a considerable slice of the state space is left unverified.

But dynamic analysis, when applied to concurrent programs, will yield an even more incomplete result set, as an error that only happens given a very specific interleaving, with a very low probability or under very specific scenarios (e.g., the *probe effect* [71, 145]), may be extremely hard, or even impossible, to recreate or even spot prematurely [145].

With such trade-offs for each analysis technique, as none can fully address every type of error [200], ideally one should combine both techniques into a hybrid approach or, at the very least, use both in conjunction in order to yield a more complete set of results [4]. Ball [12] confirmed this principle, when he argued that dynamic and static analyses are in fact complementary techniques.

2.2.1.3 Symbolic Execution Analysis

Although some authors consider it as a form of static analysis [192], symbolic execution is a form of automated concurrent software verification with a different approach than either that of static or dynamic analysis. A symbolical execution consists of analysing the correctness of a program by simulating its behaviour, using algebraic symbols, that can represent any arbitrary value, as its input, instead of actual data, like in dynamic analysis, discussed in Section 2.2.1.2. For it to become possible, computational definitions of a programming language semantics need to be extended [109] in order to allow for these symbols, which are inputted into the simulated execution, to be derived into symbolic formulae that can precisely describe the output state of a program in relation to its inputs, which represent all the possible interleavings [56, 79, 108].

These program statements along with their produced state can be broken down into a tree structure, so they can then be verified to ensure certain program properties are not violated [11, 109]. Unfortunately, with larger programs, it becomes increasingly complex to deal with exponentially rising number of computed possible program states — a concept described as *path explosion* [79]. An execution tree can even be infinite, if a program contains an infinite loop [109].

A verification condition (VC) is a logical formula, generated by a VC generator algorithm, that can be used to infer program correctness of a whole program or fragments [70]. A program can be considered correct when all its deduced interleavings can satisfy a specification or, in other words, the VCs can be shown to be theorems [56]. Symbolic execution becomes especially useful for detecting edge-case scenarios such as division by zero or null pointer dereferencing, but it can also be extended to detect more complex correctness violations [11].

2.2.2 Concurrency Error Detection

Concurrency error detection is a big and complex field of study and extensively describing it would be worthy of an entire dissertation in it of itself. As such, the following sections will merely focus on exploring a well-established tool or method that may be used in order to detect the concurrent software errors previously mentioned in Section 2.1.3, while also focusing on the type of software analysis it relies on, from the ones previously presented. In short, dynamic analysis solutions sacrifice performance and scalability to ensure a lack of false-positives [104]. But, as mentioned in Section 2.2.1.2, both techniques can be used together, leveraging each other's shortcomings. Some of the following tools can even serve more than one purpose, being able to detect a plethora of errors.

2.2.2.1 Data Race Detection

Most data race detection methods fall into two categories — *happened-before-based* (HB-Based) and *lockset-based* [193]. The first relies on the *happens-before* method by Lamport, which can be seen in Figure 2.2, while the latter was first presented by Savage in the form of *Eraser* [173], a tool created by researchers at the Universities of Washington and California at Berkeley, with the purpose of detecting data races in concurrent programs through dynamic analysis. Other methods for

atomicity and/or order violation detection include: [33, 141, 176] for dynamic analysis; [18, 104, 161] for static analysis; and [138] for symbolic execution analysis.

Eraser uses the *Lockset* algorithm to detect data races, which works by having each thread tracking its own set of currently help locks, while the program is running. In enforcing nearly all shared variables to be protected by a lock, *Eraser* can detect which reads and writes can cause a data race. Since it cannot guess which locks are intended to protect which variables, it infers this knowledge from the execution history.

Algorithm 2.1 The *Lockset* algorithm (adapted from [173]).

```
1: H_t ← [ ]                                   ▷ Map of locks held by each thread
2: C_v ← [ ]                                   ▷ Map of candidate locks for each variable

3: for all t ∈ ALL_THREADS do
4:     H_t ← { }

5: for all v ∈ ALL_VARIABLES do
6:     C_v ← ALL_LOCKS

7: function AcquireLock(t, l)
8:     H_t ← H_t ∪ l

9: function AccessVariable(t, v)
10:    C_v ← C_v ∩ H_t
11:    if C_v = ∅ then
12:        IssueWarning(v)                     ▷ Data race detected
```

The process shown in Algorithm 2.1 is called "lockset refinement" [173] and ensures that, when attempting to access a variable v, a thread t holds at least one lock l that can consistently protect concurrent accesses to v. In the event that a warning is issued, one can deduce that a data race exists because there is no lock l, currently held by a thread t that can protect a variable v.

Improvements that can be made to the algorithm include:

- Accounting for the fact shared variables are usually initialized without the need for a lock for their initialization;
- Ignoring variables which are only written to once and only read from then on;
- Allowing for special locks which allow multiple readers but only a single writer.

2.2.2.2 High-Level Data Race Detection

Atomicity and Order Violation Detection

Notable methods for atomicity and/or order violation detection include: [36, 66, 135, 156] for dynamic analysis; [192, 194] for symbolic execution analysis; and [34, 157] which uses a mixed approach. From these notable tools, *ConMem* [204] was chosen to be described. *ConMem* is a tool which can detect both atomicity and order violations in concurrent programs, developed by researchers at the University of Wisconsin. It analyses a program's run time information to find concurrency bugs, including null assignments, dangling pointers or shared pointer dereferencing.

These detected errors are all caused by a common error-propagation pattern that was inferred from studying a series of concurrency bugs present in real world programs.

Instead of analysing interleavings individually, *ConMem* is effect-oriented, only examining those that may result in a concurrency error, tackling concurrent software's natural non-determinism. Additionally, *ConMem* makes use of a noise injection module (*ConMem-v*), to further validate these errors, by finding incorrect interleavings and rooting out false positives. *ConMem* can be divided into four submodules (besides *ConMem-v*), which are used independently in order to detect a plethora of concurrency errors:

Con-NULL Responsible for locating null pointer dereference errors by resorting to a tool developed by Intel named *PIN* [166], to allow for binary instrumentation and off-line trace analysis (post-execution). More specifically, analysing thread identifiers, the program counter, memory locations, and store-value information at run time.

Con-UnInit With the purpose of finding uninitialized memory reads, caused by incorrect interleavings, at run time, also through *PIN*, albeit with a simpler algorithm that maintains per-thread information about which memory locations are initialized.

Con-Dangling Which scans the program for dangling pointers, relying on binary instrumentation through *PIN* as well. It provides a run time bug detector that identifies all accesses to memory locations de-allocated by another thread and then analyses order synchronizations to infer whether these accesses were concurrent with the de-allocation operation.

Con-Overflow Whose implementation relies upon the *Lockset* algorithm from *Eraser* [173], discussed in 2.2.2.1, to detect buffer overflows.

Stale Value Detection

An algorithm for detecting stale value concurrency errors was developed by Artho et al. [6] at Eidgenössische Technische Hochschule (Swiss Federal Institute of Technology) (ETH) and the National Aeronautics and Space Administration (NASA). It was built upon *JNuke* [7], a framework written in C, by Artho et al., that has a collection of program analysis tools for Java bytecode, allowing for verification and model checking of Java programs. Other notable methods for detecting stale value errors in concurrent software include: [76] for dynamic analysis; [31, 54] for static analysis; and [4] for a mixed approach.

The algorithm does static analysis of the program's source code without any need for assumptions or annotations. Working at the bytecode level brings the added advantage of being able to fully analyse the programs semantics and not higher-level abstractions, in order to detect stale value errors. The algorithm itself is data-flow-based, meaning it verifies the interactions between each program's variable definitions and uses in selected program paths [183].

Stale value errors, explained in Section 2.1.3.2 occur when the value of a thread's local copy of a shared variable becomes out of sync with its new value, after another thread has modified it, making the value inconsistent with the global program state. The algorithm locates instances of shared data accesses where the thread is not operating over the current values.

Definition 1 (Formal definition of stale value absence).

$$\forall i, r : r \in U_i \wedge \ B_r \in S \rightarrow B_r = B_c$$

- i - *instruction*
- r - *register*
- U_i - *set of registers used by an instruction* i
- B_r - *monitor block for a register* r
- B_c - *currently used monitor block*
- S - *set of shared monitor blocks*

Note: adapted from [6].

A monitor block is a set of instructions between a lock acquisition and a release, with a unique identifier. At the start of an execution, no registers are considered shared, and this remains true for each one until they contain the value of a shared field. So, a shared registered value can be considered stale if it originated from a monitor block than the one it was used in. This property is named *block level atomicity* [6] and can be observed formally in Definition 1.

2.2.2.3 Deadlock and Livelock Detection

Deadlock detection is a broad field, with some notable methods including: [15, 113, 131] for dynamic analysis; [142, 149, 196] for static analysis; and [19, 138] for symbolic execution analysis. A notable solution for this problem, which was chosen as the one to be described, can be found on a paper by Demartini et al. [53], describing a deadlock detection method for concurrent Java programs.

Its abstract formal model is expressed in Protocol Meta Language (PROMELA) [90], a language used to describe software, by specifying their behaviour in formal validation models. For checking these models, another model checking tool named *SPIN* [89], created by the same author of PROMELA, is used. *SPIN* is a generic verification system for designing and verifying concurrent systems and built to interpret specifications written in PROMELA. The author of the deadlock detection method revealed that the choices for using PROMELA and *SPIN* were mainly due to former's flexibility and the latter's efficiency [53].

This deadlock detection method consists of three steps, with the first being the translation of the initial source code of the software into PROMELA, resorting to a purpose-built tool, created by the same researchers for this study, named *Java2Spin*. Afterwards, an analysis is performed on the translated model with *SPIN*, with the purpose of checking for any program states that can lead to a deadlock. Lastly, any sequences of program states that may lead to an error are converted back to Java and reported back to the user, allowing him to debug the faulty code.

Most livelock detection methods use model checking — a powerful verification technique that allows for an efficient exploration of the state space, effectively tackling the state space explosion problem [38, 58, 104]. Tai presented a paper describing a model checking method [185] for livelock detection. Some other examples are [78, 117] for model checking; and [130] for dynamic analysis. Model checking has the unfortunate drawback of scaling poorly as systems grow in size and complexity [63].

The algorithm proposed by Tai works by searching a constructed reachability graph for a program — a directed graph containing the program states and their transitions. The first step of the method is transforming said generated graph into a condensed acyclic tree structure, allowing the algorithm to traverse it and detect states that can lead to a livelock.

Algorithm 2.2 Livelock Detection Algorithm (adapted from [185]).

```
1:  CRG_P ← GenerateCRG(P)          ▷ Condensed Reachability Graph for a program p
2:  B_n                             ▷ Map of processes busy waiting[1] for a node n
3:  L_n ← { }                       ▷ Set of processes livelocked in node n

4:  for all n ∈ CRG_P do            ▷ Traverse each node n, ideally through a DFS
5:      if n.IsLeaf() then
6:          L_n ← B_n
7:      else
8:          L_n ← B_n ∩ GetChildrenBusyWaitSet(n)

9:  function GetChildrenBusyWaitSet(n)
10:     B'_n ← { }
11:     if n.LeftChild() ≠ ∅ then
12:         B'_n ← B_n ∪ n.GetChildrenBusyWaitSet(n.LeftChild())
13:     if n.RightChild() ≠ ∅ then
14:         B'_n ← B_n ∪ n.GetChildrenBusyWaitSet(n.RightChild())
          return B'_n
```

[1]Not deadlocked, terminated, or executing a progress statement in any state in n

Algorithm 2.2 shows an adapted version of the original livelock detection algorithm. A livelock is said to be present in a program P, if Condition 1 is verifiable.

Condition 1 (Livelock formal condition).

$$\exists p \in P \; \forall n \in CRG_P : p \in L_n$$

Note: adapted from [185].

2.2.2.4 Starvation Detection

In Section 2.2.2.3, an algorithm for livelock detection was described. In the same paper, Tai [185] proposed a model checking method with the intent to detect starvation in concurrent programs. This method will be described throughout this section. Its algorithm is built upon the other two solutions proposed, for livelocks and deadlocks, as it needs to also detect these two, in order to properly work. Other examples of model checking for starvation detection are [94, 178].

The algorithm itself, uses the same principle as its livelock detection counterpart, making use of the same condensed reachability graph containing all states and transitions for a given program P. A "fair cycle" is a cycle in the graph, prior to being condensed, in which every process $p \in P$ either contains a state transition or is blocked or terminated in every state. Ultimately, for a program P to be starvation free, it cannot reach a state in a fair cycle in which it cannot progress, even though it is not deadlocked, livelocked or terminated. Another important concept for understanding the algorithm, is that of the "no-progress" cycle. A cycle is said to be a "no-progress" cycle for a given process $p \in P$, if p cannot make progress in any state of this cycle.

Algorithm 2.3 Starvation Detection Algorithm (adapted from [185]).

1: $CRG_P \leftarrow GenerateCRG(P)$ ▷ Condensed Reachability Graph for a program p
2: $S_n, D_n, L_n \leftarrow \{\ \}$ ▷ Set of processes starving, deadlocked and livelocked in node n

3: **for all** $n \in CRG_P$ **do** ▷ Traverse each node n, ideally through a DFS
4: $F_n \leftarrow CRG_P$.SearchFairCycles(n) ▷ Compute map of fair cycles for a node n
5: $T_n \leftarrow CRG_P$.GetTerminatedProcesses(n) ▷ Get set of processes terminating at node n
6: $NP'_n \leftarrow \{\ \}$ ▷ Set of processes for which node n contains a fair, no-progress cycle

7: **for all** $c \in F_n$: c.IsNoProgress() **do**
8: $NP'_n \leftarrow NP'_n \cup (c$.GetProcesses() $\setminus T_n)$

9: **if** $F_n = \emptyset$ **then**
10: $S_n \leftarrow \{\ \}$
11: **else**
12: $NP_n \leftarrow$ ComputeNoProgressProcesses(F_n, T_n)
13: $S_n \leftarrow (NP_n \setminus D_n) \setminus L_n$

An adapted version of the original starvation detection algorithm can be found in Algorithm 2.3. A program is said to be starving if Condition 2 is valid.

Condition 2 (Starvation formal condition).

$$\exists p \in P\ \forall n \in CRG_P : p \in S_n$$

Note: adapted from [185].

Chapter 3

Related Work

One of the key takeaways from Chapter 2, is how concurrent software is inherently non-deterministic by nature and, as such, extremely difficult to debug. More importantly, some bugs are only manifested through very specific interleavings, which may have a low probability of occurring. These bugs are more affectionately known in the software world as *heisenbugs* [74], an homage to Heisenberg's Uncertainty Principle, given their characteristic unpredictability. For the sake of rooting out these various elusive bugs, researchers have developed the concurrent software technique known as "noise injection". Noise injection becomes all the more important, given that conventional testing methods may have a hard time running into the interleavings that result in these bugs: unit-testing is inherently too limited, as it only manipulates the program input and concurrent software is also dependent on temporal events; stress testing a concurrent program with a high number of threads is not a very effective approach in it of itself [73]; and, lastly, a systematic and exhaustive state space exploration approach (as proposed in Java PathFinder [191]), may actually only cover a very limited number of interleavings, rendering the strategy infeasible [63]. Additionally, differences between multiple runs, when using conventional testing methods, are almost solely due to input/output (I/O) delays and network load, meaning they may not be repeatable at all, making it extremely hard for the programmer to find the original cause for the error [58].

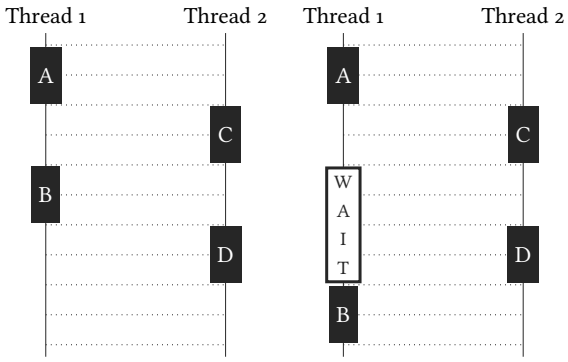
3.1 Noise Injection for Concurrent Software Testing

The rationale behind noise injection is the *probe effect* [71], which describes how seemingly inexistent concurrent software errors can manifest themselves, as delays are introduced into a program. In short, noise injection has the potential to increase the state space coverage of a test, by forcing the triggering of additional interleavings and causing atomic blocks to be executed for a longer timing period, significantly increasing the probability of encountering a state which violates the program's invariants (i.e., an error or bug) [63, 128]. This effect can be visualized in Figure 3.1, with noise being injected into a thread, forcing it to wait, resulting in an alternative interleaving. These errors, although present, remain undetected due to being substantially "far"

from each other in terms of execution time, with noise being the mechanism able to make them coincide in a time frame.

Research has shown that noise injection testing of concurrent software can greatly increase the overall probability of finding errors, by forcing different interleavings [63]. Additionally, noise injection, when used in conjunction with other methods, can potentially show a significant increase in the probability of triggering interleavings that lead to concurrency errors [128], further increasing the method's effectiveness.

It is important to note, however, that noise injection is, by its nature, a confidence technique, as opposed to model checking, for example, seen in Section 2.2.2.3, which is a correctness technique. This essentially means that noise injection cannot guarantee to trigger every interleaving, covering the whole state space. Instead, noise injection is a best-effort technique, in which, most of the time, coverage of the state space is significantly increased.



(a) Without noise injection (b) With noise injection

Figure 3.1: How noise injection can force different interleavings.

3.1.1 Noise Types

When injecting noise into a concurrent program, it is vital to choose the correct type or types of noise to inject. Along with the chosen noise type, it is of the utmost importance to take into account the intensity of the noise. Many solutions, such as [58, 128], rely on randomizing the duration of *sleep* statements every run, with the intent of triggering different interleavings.

Many of the noise injection primitives rely on interrupting the threads' normal behaviour by, for example, forcing them to wait. This section will focus on implementations for the Java programming language and Portable Operating System Interface (POSIX) implementations of the C/C++ programming languages. Notable examples for these languages include:

- **Sleep**: Inserting a `Thread.sleep(t)`, for Java, or `sleep(t)`, for C/C++, statement at a location, which temporary suspends the execution of the current thread's routine for an interval of time `t` which is passed as a parameter. This method is included both in Java and C/C++, with the latter being implemented by the operating system. Previous research in noise injection has verified that the `sleep` primitive is, overall, the most effective noise

type, resulting in more effective state space exploration and, consequently, improved error detection [58];

- **Yield**: Inserting a Java-specific `Thread.yield()`, or the analogous `pthread_yield()`, statement at a location which will hint the scheduler of a thread's willingness to yield its current use of a processor. The effect of this operation is non-deterministic, as the scheduler is free to ignore it. The intensity of the `yield` noise type is expressed as the total number of times the statement is invoked [63]. It is important to note that, if there are more physical threads than program threads, the `yield` noise type will have basically no effect, except for the delay added by the method invocation and subsequent verification, as the scheduling cannot be affected;

- **Wait**: Inserting either the Java-specific statement `lock.wait()` at a location, or the C/C++ analogous `pthread_cond_timedwait(...)`, exhibits a very similar behaviour to the aforementioned *sleep* primitive, with the particularity that it requires the thread to release any lock or monitor it is currently holding. *BusyWait*, a notable noise injection heuristic which relies on this noise type, keeps the thread looping for a given amount of time, instead of obtaining a monitor. This heuristic has been shown to provide good results, but at the cost of efficiency, as it can be expected to increase the overall program run time considerably [63, 202];

- **Priority**: Inserting Java-specific `Thread.priority(p)` statements at a location, which manipulates a thread's priority, by setting it to the passed parameter `p`. The threads with a higher priority value will be executed in preference to the ones with a lower value. It is also possible to manipulate scheduler priority in POSIX systems through C/C++, although it is a more complex endeavour, requiring additional configuration. Another useful technique involving priority primitives, is leverage the scheduler's power in order to achieve laziness or eagerness, either by lowering or increasing a single thread's priority, relative to all others', respectively. While this can be straightforward in a single processor system, it may become increasingly more complex or even outright unachievable in a multiprocessor system, depending on the scheduler that is being used;

- **Suspend**: Inserting a `Thread.suspend()`, for Java, or the analogous C/C++ `pause()` statement at a location, which suspends a thread indefinitely until another thread wakes it up through a `Thread.resume()` or `signal(signum)` statement. This could also be done based on a specific condition, defined by an heuristic. This method is currently officially deprecated for Java, as it is inherently dangerous for easily leading to deadlocks [57].

However, the use of any of these noise injection primitives will result in some level of performance degradation, attributed to the additional thread delays and additional context switching which is forced upon the scheduler [58], even in cases where a *sleep* has a nil value, or a *yield* is ignored by the scheduler.

3.1.2 Noise Injection Heuristics

As a program's length increases, so do the number of possible interleavings, exponentially increasing the state space, relative to the number of instructions and threads. Utilizing noise heuristics allows for the tackling of the state space explosion problem, allowing for the generation of more, and hopefully more relevant, interleavings without having to explore the entire exponentially growing state space [58]. Relying on a brute-force noise injection approach, which tests all possible interleavings by inserting noise at every single program location and then running all possible combinations, although technically correct as a noise injection technique, is extremely inefficient [63], due to this exponential growth quickly becoming unmanageable. Noise injection heuristics present themselves as a solution to this problem by defining methods that attempt to precisely noise the locations which are the most likely to trigger different interleavings, therefore raising the probability of triggering an interleaving that violates the program invariants. In summary, the main purpose of heuristics is to define methods which are more likely to disturb the scheduling [65]. As expected, not all heuristics are equally efficient and effective at generating different interleavings, which represents some of the biggest challenges in the field of noise injection — continuously improving existing heuristics, creating novel heuristics and, finally, finding the right heuristic for a given application, as a one-size-fits-all heuristic does not exist [126].

Noise injection heuristics are made up of two components — noise placement and noise seeding —, which together define where to place noise and how it is to be generated [63]. More precisely, noise placement is responsible for defining which types of program locations are the most suitable for noising (i.e., most likely to affect scheduling), along with when (i.e., in which runs of the program) the noise at these locations will be triggered [126]. Choosing to insert before and after accesses to atomic blocks, would be a perfectly valid and sensible choice for a noise placement heuristic. As can be expected, inserting noise at one point of the program, always has an effect, from extremely high to extremely low, on the entirety of subsequent instructions. As such, it is no surprise that, inserting noise to influence the scheduling at a specific location, can actually influence, albeit with a lower probability, the scheduling at another unnoised location.

Noise seeding, the other component of heuristics, includes choosing a noise type to be generated (e.g., from the ones seen earlier in Section 3.1.1), its frequency and its intensity [93]. Frequency is defined as the probability function of noise being triggered at each of the previously chosen noising locations. Intensity is characterized by how much the system itself is impacted by the noise, a concept some authors call "strength" [63]. Together, frequency and intensity are essential for triggering different interleavings. Manipulating frequency through the use of probability functions is an extremely effective and widely used technique found in [58] and [116]. Intensity can also discover new and different interleavings through the stressing of systems resources, the scheduler or the message-passing interface (MPI).

Listing 3.1, shows an example of a noise injection heuristic in a program's instrumented source code. The `sleep()` statement defines the noise type of this heuristic, with its parameter defining an interval which is the intensity of this noise. Lastly, the probability function `rand`, defines a 50% chance of noise occurrence, being indicative of the noise frequency.

```
void update_last_client(int age, int nationality, ...) {
    pthread_mutex_lock(&mutex);
    client.nationality = nationality;
    pthread_mutex_unlock(&mutex);

    if (rand() % 2 == 1)
        sleep(TEN_MINUTES);

    pthread_mutex_lock(&mutex);
    client.age = age;
    pthread_mutex_unlock(&mutex);
}
```

Listing 3.1: Real-world noise injection type example using *sleep*.

Listing 3.1, also allows us to visualize how adding noise can greatly increase the chances of detecting high-level data races. As proof, one can imagine a scenario of a store that takes an average interval of five minutes to serve each client. This scenario will probably almost never trigger the interleaving that causes the high-level data race, because both the atomic blocks will immediately execute. The process of injecting noise through the `sleep()` statement enables the system to reach an interleaving that can effectively corrupt the client data. Another important takeaway from this example is that the sleep time had to be, at the very least, five minutes, or else no conflict would exist in this program. This means that the noise duration is as important to define as is its location.

Together, noise placement and seeding define heuristics that have the potential to highly increase the probability that every time a test is run, it can explore more of the program's search space, triggering new and different interleavings.

Figure 3.2 illustrates how different noise seeding and placement heuristics can affect a program's routine, causing different interleavings. Comparing, from left to right, the first and the second picture, the noise intensity remains the same, but the placement is changed. Between the first and third picture, the placement remains the same, but the intensity has changed.

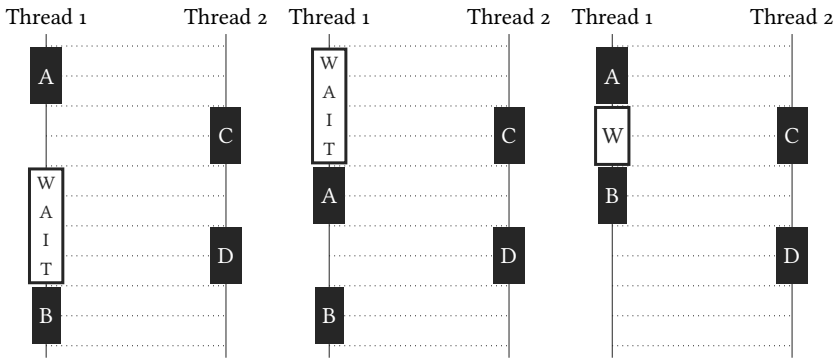


Figure 3.2: How different noise injection heuristics can trigger different interleavings.

Since the inception of noise injection as a field of study, researchers have proposed a plethora of different noise injection heuristics. As part of this dissertation's research, a taxonomy for these

heuristics was developed, with the express purpose of categorizing them in a more concise and higher-level manner:

- **Synchronization-block-based**: For this heuristic, noise can be injected before, after and/or inside synchronized blocks. This heuristic is commonly used for noising locations that are previously already suspected of being problematic. A common and efficient noise placement heuristic is to insert noise before and after these synchronized blocks [63], such as in *ConTest* [58]. Still, it is also interesting to insert noise inside these blocks, increasing their duration, and thus their probability of triggering an erroneous interleaving;

- **Read/Write-based**: In this heuristic, noise is injected before and/or after shared-variable accesses. This heuristic is particularly helpful for finding common write-read and read-write data race scenarios [63]. This heuristic can be found in notable noise injection tools, such as *ConTest* [58]. Fiedor and Vojnar [65] experimented with varying noise seeding, more specifically frequency, before and after the accesses. This technique is then further extended by varying frequency in regard to the type of access (i.e., read or write) [63]. Another interesting option, is to inject noise into global variables, as seen on the work by Garcia [72], as these have a high probability of being accessed by multiple threads. An algorithm can be used to detect shared variables, so that only the accesses to these variables are tracked and noised. Since shared variables are registered, this heuristic can be used while focusing the testing process on specific, previously suspicious variables. Research has shown that restricting noised locations to shared variable accesses will increase error detection probability [62];

- **Function-based**: Insert noise either inside of, before and/or after function calls. This technique can also be used to noise specific monitored functions;

- **Thread-based**: Insert noise that is only triggered for a specific thread. This may be used to implement the "HaltOneThread" [187] seeding technique, which occasionally stops one thread with the intent of verifying the moment up to which the other threads cannot progress anymore. "InverseNoise" is a variation of this technique, whose strategy is its exact opposite, instead occasionally stopping all threads but one [63];

- **Pattern-based**: In this heuristic, noise is injected in program locations where the code exhibits a particular pattern, such as repeated accesses to the same variables in a method, as proposed by Fiedor et al. [63], with the intent of finding atomicity violations. The authors also encourage the noising of variables which are contained in atomic blocks, in order to validate the integrity of their atomicity properties. Additionally, this heuristic can be used via algorithms that detect suspicious (i.e., error-prone) code blocks;

- **Random**: A mix of all the other heuristics, with noise being triggered randomly when arriving at locations relative to each heuristic. Fiedor et al. [63] mentioned a technique, used by *ConTest* [58], which also randomizes the noise seeding heuristic to be triggered at

each location. However, this last technique has in some cases shown to perform poorly, actually masking concurrency errors [114];

It is important to keep in mind that this is not an exhaustive list and that none of these noise localization and seeding heuristics are mutually exclusive, allowing the use of a hybrid solution that employs a harmonious mixture of various localization and seeding heuristics. The intent of this list is merely to categorize into an easy-to-read format, all the different approaches to noise injection which were found while researching the state of the art in noise injection. Lastly, research has shown that no heuristic is best for every use case, with every situation requiring the tester's careful consideration [127]. This task has been formalized as the TNCS problem [93], which has been put into practice in *CFLASH* [72].

Additionally, none of these noise localization and seeding heuristics are mutually exclusive, allowing the use of a hybrid solution that employs a harmonious mixture of various localization and seeding heuristics. Although all noising techniques essentially have the same purpose, to influence the scheduler to disturb a running program's behaviour, some are more effective at this task than others [58]. Furthermore, many noise heuristics rely on a specific type of noise for their implementation, meaning noise types are not interchangeable, with an example being the Barrier heuristic proposed in [62]. However, the use of any of these noise injection primitives will result in some level of performance degradation, attributed to the additional thread delays and context switching is forced upon the scheduler, even in cases where a *sleep* has a nil value, or a *yield* is ignored by the scheduler.

Lastly, research has shown that no heuristic is fit for every use case, with every situation requiring the tester's careful consideration, when choosing and configuring a noise injection heuristic [127]. This task has been formalized as the test and noise configuration search (TNCS) problem [93], which has been put into practice in a tool called *CFLASH* [72].

3.2 Static vs. Dynamic Noise Injection

Noise injection techniques also need to be tailored to the specific concurrency software model of the software that is to be tested. While techniques such as manipulating the scheduling make sense for shared-memory programs, message-passing programs will need to introduce this noise at a network-level through their communication medium.

Noise injection is a testing technique and, as such, can also be broken down into its respective static and dynamic variants. But, noise injection is, in it of itself, a dynamic analysis technique, due to the nature of how it works — it needs a program to be running to work, as noise is always triggered during run time. As such, the "static" and "dynamic" terms, in noise injection, only describe how the noising instructions are inserted into the program — before or during run time —, making these categories not directly related to the generalized definitions of software analysis types we've looked at, back in Section 2.2.1. Essentially, static noise injection relies on static instrumentation to insert additional code and data into a program's source code or executable, generating a new, modified executable. Similarly, whereas dynamic noise injection relies on

dynamic instrumentation to insert additional non-persistent code and data during a program's execution [122].

Early noise injection testing relied on a manual insertion of statements that affect scheduling (e.g., sleep, yield) at key locations in code [21], a process which yielded positive results in research by Brat et al. [22]. However, this method quickly proves itself inefficient, as inserting statements manually for any moderately complex concurrent program becomes extremely time-consuming. As such, researchers rely on tools which automate the insertion of noise triggering statements into key locations, based on noise injection heuristics.

The next two subsections will take a deeper dive at the specifics of static and dynamic noise injection tools for testing shared-memory and message-passing software. The following subsection will present established tools for instrumentation, which can be used for static noise injection.

3.2.1 Static Noise Injection

For the shared-memory model, static noise injection tools inject noise into a program's source code, intermediate code (e.g., bytecode/opcode, for Java or .NET) or, through binary instrumentation, binary code (e.g., for C/C++ programs), in order to generate new, noised executables. Opting for lower or higher-level noise injection tool is always a compromise between implementation difficulty and precision [63], as lower-level instrumentation allows for a more fine-grained code manipulation. However, binary instrumetation has the perk of being able to be generalized, making it more easily and broadly applicable, due to the lack of needing any prior knowledge of its source [151, 203]. This is especially true in Java, since its concurrency primitives are very high-level, as they are effectively three layers above actual native machine code.

Source analysis has the perk of being platform-independent, more easily portable and more easily maintainable, as they do not need custom compilers or interpreters. In *ConTest* [58], these are mentioned as the main reasons for opting for static noise injection. One of the biggest perks of resorting to static instrumentation is that, since all the instrumentation is done before the program is run, the overhead is minimal when compared to techniques resorting to dynamic instrumentation [17]. However, this strength comes as a double-edged sword, since any dynamically loaded classes cannot be included in its instrumentation routine.

Currently, no tools exist for injecting noise statically into message-passing software, with available tools relying instead on dynamic noise injection, which is usually done via the IPC channel. As such, noise injection into message-passing software will be discussed later, during Section 3.2.2, when discussing dynamic noise injection tools.

The next few sections will be dedicated to the presentation of notable examples of static instrumentation tools for source, intermediate and binary code manipulation. Although, none of these tools were created with noise injection specifically in mind, they may be used in order to instrument noise into a concurrent program's source. It would be wise of a noise injection tool developer to make use of an already established instrumentation tool to achieve static noise injection whenever possible, as opposed to undertake the added challenge of creating a novel solution specifically for this end.

3.2.1.1 TXL

TXL [41, 42, 43], is a special-purpose programming language created by Cordy et al., a researcher from Queen's University in Canada. Its main interest, in the area of noise injection, is the fact that it provides an approachable syntax and efficient algorithm for the manipulation of software's source code (i.e., source-to-source transformation).

TXL solutions consist of grammars that need to be written by a programmer for specific programming languages and libraries/frameworks. These grammars are then fed the source code of programs that one wishes to modify, outputting the respective modified source. The original source code symbols are interpreted as tokens and separated into two kinds of lists — choice lists and order lists. The first list represents alternation, such as conditional statements, and, the latter, instructions that are executed sequentially. The resulting data structure is a parse tree that is both fast and easy to navigate, all the while following the original program's logical structure. A programmer can then create transformation rules, by specifying patterns to be matched and replacements of code to be inserted.

An example of a TXL parsing program can be found in Listing 3.2.

```
include "C.Grm"

define program
    [expression]
end define

define expression
    [number]
    | [expression] = [number]
    | [expression] == [number]
end define

rule replaceCurrentYear
     replace [number]
        2021
     by
        2022
end rule
```

Listing 3.2: Basic TXL example — update instances where previous year was used.

In short, TXL defines a grammar, parses a program's source code into a tree, transforms it through some specified rules, and unparses the tree into a new, modified source code. Another perk of TXL is the fact that it conveniently comes packaged with base grammars for commonly used programming languages, such as Java, C, C++ or Pascal.

TXL can therefore be used to statically inject noise into software, by means of finding critical sections via its tokenization procedure. An example of such a solution is Concurrency Faults Localized Automatically using Search Heuristics (CFLASH) [72], developed by Garcia at Ontario Tech University, a tool which was built with the purpose to automatically detect and localize

concurrency faults in concurrent Java software. It uses TXL to parse the given Java program and injects `Thread.sleep()` statements, with varying intensity, before synchronization operations.

Other notable examples of source instrumentation tools include *DMS* [14] and the *ASF+SDF* Meta-Environment [190].

3.2.1.2 Soot

Soot [188, 189] is a framework for Java presented in 1999 and finally released in 2000. It started out specifically with the goal of allowing for bytecode (intermediate) code optimization and has since evolved into a fully-fledged Java framework tool that can serve multiple additional purposes for both Java source code and bytecode (intermediate) code, with instrumentation being the most important feature, concerning the field of noise injection in concurrent software. It currently supports both Android and Java bytecode and Java source code up to Java 7 and can translate any of these to another. Java 9 source code compatibility is currently under development and as a functionality, it is advertised as *bleeding edge*.

Soot is essentially a compiler, which transforms Java source or bytecode into other formats [118]. Additional functionality is implemented via extending the compiler with additional steps which may analyse and/or transform the code. It uses an intermediate representation language — called *Jimple* —, to abstract the Java bytecode, simplifying its analysis and instrumentation. Additionally, Soot supports mechanisms that provide *Jimple* with the means to transform its input into a plethora of different outputs, such as: transformed class files, error messages, HTML or graphs describing analysis results and creating (potentially transformed) class files annotated with results obtained from program analysis [118]. *Soot* was designed with extensibility in mind, allowing a programmer to extend its functionality through methods that are executed on compiler passes called *transformers*.

The *Soot* developers made their tool completely free and open-source, by releasing it under a GNU General Public License (GNU GPL) licence. This allowed dozens of institutions worldwide, like universities and research centres [81], to freely use it for educational and scientific purposes. Not only is *Soot* still actively maintained, it also has the perk of being extensively documented on their main website.

Through its instrumentation functionality, *Soot* is a powerful tool which can be used for applying noise injection to a Java program at a bytecode-level.

Other notable examples of bytecode instrumentation tools include *ASM* [29], *BCEL* [47] and *JOIE* [39].

3.2.1.3 PEBIL

PMaC's Efficient Binary Instrumentation Toolkit for Linux (PEBIL) [122] is a static binary instrumentation tool for Linux, developed by researchers at the San Diego Supercomputer Center at the University of California. It is completely free and open-source, although not actively maintained

anymore. Additionally, it supports the instrumentation of Linux's Executable and Linkable Format (ELF) files for x86 and x86-64 architectures, containing not only native code, but also C/C++ concurrency libraries like *OpenMP* [46] and POSIX Threads (pthreads).

PEBIL was created primarily with efficiency in mind, becoming a more lightweight alternative to other static binary instrumentation tools. Its instrumentation process works by inserting branch instructions at multiple locations in the code, that transfer the application's control to the added instrumented instructions. This process is devoid of side effects, as it starts by first saving the program state, then transferring control to and executing its instrumented procedure and lastly, after the procedure has completed, it proceeds to restore the previous program state, allowing the instrumentation to occur in isolation from the application code. PEBIL requires this process to ensure another important functionality — the insertion and initialization of data to be processed within the instrumented sections.

Although an exponentially more complex approach, PEBIL allows for an incredibly fine-grained approach to the injection of noise into concurrent software. This becomes extremely rewarding, as it eliminates the need for creating different tools and implementations/configurations for every single programming language and, additionally, does not require any of the source code to be known beforehand.

Other notable examples of binary instrumentation tools include *Vulcan* [180] and *IDA2Obj* [99].

3.2.2 Dynamic Noise Injection

Dynamic noise injection describes methods which rely on dynamic instrumentation, or other analysis techniques, of inserting noise into a program, while it is running. It differs from static noise injection, due to not needing any prior knowledge of the program's code. A common way dynamic noise injection can be achieved, for software written resorting to the shared-memory paradigm, is by relying on a dynamic instrumentation tool, which provides an abstraction layer with the ability to capture instructions before they are issued to the scheduler. Dynamic instrumentation requires that its instrumentation process be done at a layer immediately before the scheduling. As such, dynamic source instrumentation is not possible, since even interpreters themselves tend to rely on intermediate languages (e.g., Java, .NET, Python) that is fed into their respective virtual machines.

Running a noise injection tool, relying on dynamic instrumentation, comes with a considerable downside — a very significant slowdown [122], by a factor of 100 or more (as witnessed by Fiedor and Vojnar [64] during research) while running the program and analysing it, severely affecting the overall running times. Choosing static over dynamic instrumentation requires leveraging the former's efficiency and flexibility, and the latter's coverage, as dynamic instrumentation allows for self-modifying and self-generating code [65]. This is not just a performance penalty, as the programmer must take into account how the added overhead of dynamic instrumentation can influence measurements being taken or jeopardize ongoing test routines [17]. For example, working at the virtual machine level and effectively creating a custom implementation, has the

advantage of allowing scheduler manipulation, with the added risk of updates to the interpreter not being compatible with these implementations [58].

Still, dynamic code instrumentation comes with a signifficant perk that can, in many cases, outweigh the performance penalties incurred from using it — the possibility to instrument every single class, even those not yet loaded [17].

Similarly to how Section 3.2.1, which described static noise injection, was structured, the next few sections will be dedicated to the presentation of notable examples of dynamic instrumentation tools.

3.2.2.1 FERRARI

The Framework for Efficient Rewriting and Reification by Advanced Runtime Instrumentation (FERRARI) [17] by Binder et al., is a dynamic framework for intermediate code instrumentation for concurrent Java software, first presented in 2007. Instead of relying only on dynamic instrumentation, FERRARI resorts to a mixed approach using static instrumentation also, leveraging the strengths of both paradigms. It enables the instrumentation of the entirety of the Java Development Kit (JDK) components and any other classes loaded dynamically while the program is running.

FERRARI relies on static instrumentation, to instrument the core classes of the JDK, and dynamic instrumentation, to instrument any other classes that are loaded at run time. This approach is necessary, as the instrumentation step cannot disrupt the bootstrapping of the Java Virtual Machine (JVM), with the initial static instrumentation of the JDK being done after the conclusion of this step.

Users who wish to use FERRARI to instrument a program's bytecode, have access to a simple API to define custom instrumentation-processes. These consist in instrumentation instructions which are defined in Java files by users. During the program's run time, these instructions will then be instrumented into any loaded classes and captured instructions. This implementation for user-defined instrumentation relies on an agent with a dynamic classloader. This same classloader then resorts to *BCEL* [47], a static instrumentation library for Java developed by Apache, to instrument the Java classes' bytecode on-the-fly.

Other notable examples of dynamic intermediate code instrumentation tools include *BIT* [124] and *DiSL* [140].

3.2.2.2 PIN

PIN [166] is an open-source dynamic binary instrumentation tool developed by Reddi et al. at Intel, and first presented in 2004. It is available for IA32 and amd64 architectures under Linux, Windows and macOS, with plans to add support for ARM architectures, as well. One of its primary goals was to serve as a teaching tool for students of computer science at higher education institutions.

As a dynamic instrumentation tool, *PIN* instruments additional code into an application at run time. It comes with the added perk of having access to all the instrumented application's

instructions, libraries and data. Still, in order to guarantee it can do its instrumentation transparently to the application, it also employs a private memory stack and heap for its internal logic. As a dynamic instrumentation tool, *PIN* boasts of allowing any programmer to test it application simply by feeding its binary to it.

Applications are instrumented via C/C++ files, through which *PIN*'s API methods may be called, much like a regular library. These methods describe events, such as a program emitting its start signal, its exit signal and the program's instructions being called. The programmer can then add additional logic before or after these events are executed by the scheduler. From a noise injection perspective, it becomes obvious how this API can make the injection of code containing noise primitives, rather straightforward.

Besides *PIN*, additional examples of notable dynamic binary instrumentation tools include *Valgrind* [152, 153] and Dyninst [30, 87].

3.3 Noise Injection Tools

The purpose of this section is to present the reader with notable examples of tools relying on noise injection which were found during the course of this dissertation's research. Unlike the previously presented static and dynamic instrumentation tools, which may be used to inject noise, although this is not their only or specific purpose, these tools are specifically designed to allow programmers to validate their concurrent software implementations resorting to noise-injection-based testing.

This process will fulfil the important purpose of allowing us to compare all the existing solutions, thus obtaining invaluable insight towards the development of this dissertation's proposed tool. Following the presentation of these tools, the reader will find a table with a succinct view of their different perks and characteristics, allowing for an easy high-level comparison between them.

For clarity, throughout this document a distinction will be made between noise injection frameworks and testing tools. Noise injection frameworks give a programmer the possibility to create new testing tools on top of it, providing a few heuristics implemented and the means to control noising and seeding through some sort of API. However, a noise injection testing tool is more opinionated, and runs a specific pre-determined test with optimized heuristics to serve a given purpose (e.g., trigger data races or deadlocks).

3.3.1 ConTest

ConTest [58] is a notable noise injection framework for concurrent Java software. Before *ConTest* can effectively start testing the Java concurrent program, its source code must first go through a source-level static instrumentation step. Every assignment in the program is broken down into two separate concurrent events — *before* and *after* —, allowing control of the program to be passed on to *ConTest* when assignments are made. Then, *ConTest* modifies a Java program by

seeding locations where shared-memory accesses and synchronization events occur, based on a random or coverage-based metric. Its architecture is divided into three components:

Replay component An adapted version of the *DejaVu* Java deterministic replay algorithm from Choi and Srinivasan [37], is a mechanism that can tackle the non-deterministic aspect of concurrent software testing, where test results cannot be repeated by simply re-executing the program. This algorithm essentially adds a means of traceability to noise injection testing, allowing a tester to accurately check and reproduce the exact steps that led to the concurrent error.

Seeding component A component which heuristically seeds `Thread.sleep(t)`, `Thread.yield()` and `Thread.priority(p)` noise injection primitives in order to trigger new and different interleavings. This seeding technique was crucial, as it allowed for a dramatic increase in the probability of finding typical concurrent faults. However, these heuristics are not sufficiently effective on their own, as an exponentially growing interleaving space can mask low-probability interleavings. In order to mitigate this, *ConTest* employs code coverage analysis to control the way interleavings are generated, vastly reducing the exponentially large interleaving space to polynomial size. *ConTest* allows for noise injection before and/or after concurrency events [63].

Fault detection component A component which checks for the program's correctness on a given test.

3.3.2 ANaConDA

ANaConDA [64] is a noise injection framework for concurrent C/C++ software built by researchers at the University of Brno, in the Czech Republic, built upon the dynamic binary instrumentation tool *PIN* [166], which was presented in Section 3.2.2.2. ANaConDA was partly inspired by *ConTest* and, as such, may be viewed as its C/C++ counterpart, albeit resorting instrumenting the binary code dynamically instead of the source code statically.

The framework uses dynamic binary instrumentation of the program in memory before execution, resulting in a generic solution that can be used for any piece of software. The framework can abstract itself to support multiple multithreading libraries (e.g., C/C++ POSIX Threads), as it effectively abstracts from its implementation. This is necessary because multiple tools implement equivalent synchronization primitives, representing them differently. But, this approach has the caveat of requiring the user to manually configure the framework for the specific concurrent libraries, when writing the custom analyser for the given concurrent library. This analyser is a file containing the functions that *ANaConDA* should call when a specific synchronization event, such as a lock acquisition, occurs.

At the time of its launch, only the *sleep* and *yield* primitives were implemented and the execution overhead resulted in a slowdown of around 100 times [64].

3.3.3 AtomRace

AtomRace [128] is a low-level data race detection tool for Java software, built on top of *ConTest* (mentioned in Section 3.2.2), which relies on noise injection. It was developed by researchers at the university of Brno, in the Czech Republic. It relies on a new algorithm that can detect both of these types of errors on-the-fly, without reporting false positives. This is due to the fact that its mechanism relies on checking a program's correctness during execution, through Java bytecode instrumentation, instead of the analysed program's synchronization primitives. Although the absence of false positives is a desirable characteristic, it does come with a probability of missing some errors.

AtomRace has the particular characteristic of not only detecting errors but to also attempt to self-heal when encountering them. This is accomplished either by adding synchronization or influencing the Java scheduler, allowing for the circumvention of errors. Although not guaranteed to fix the error, the self-healing can guarantee it will not worsen or cause additional errors, as its technique relies on injecting additional locks into the application.

The architecture itself is divided into three main components, which are effectively run in a pipeline fashion. The *execution monitoring* module analyses the program on-the-fly and reports suspicious events to the *analysis engine*. The latter then runs the *AtomRace* algorithm, reports any errors found and communicates them to the *healing logic* module to attempt to self-heal from them. The algorithm itself was designed to cause an absolute minimal overhead, as the tool was created with the purpose of being deployed into production environments.

3.3.4 RaceInducer

RoadRunner [68] is an established open-source dynamic analysis framework written in Java. It provides an API that acts as a middleman between its own event stream and custom analysis tools built on top of it. Furthermore, it is possible to chain these tools together thanks to its *tool-chain* architecture, which allows the developer to take advantage of a modular design. The researchers behind *RoadRunner* claim that its highly-optimized implementation boasts comparable performance to other dynamic analysis solution, at a fraction of the code size.

RoadRunner's goal was to provide an accessible API that allowed developers of analysis tools to abstract themselves from low-level intricacies. Many tools have been built on top of *RoadRunner*, most notably race detectors such as *Goldilocks* [59], *FastTrack* [67] and *Atomizer* [66].

Recently, Yu et al., at Electronics and Telecommunications Research Institute (ETRI) in Korea, have presented *RaceInducer* [202], a novel noise injection tool built on top of *RoadRunner*. The first step of *RaceInducer* is to analyse a piece of software for possible instances of data races, resorting to *FastTrack*, a hybrid algorithm based which employs the *Lockset* algorithm, mentioned in Section 2.2.2.1, and the *happens-before*, described earlier in Section 2.1.1.2, techniques. Its second step consists in using this gathered information to noise the program during run time, with the intent of triggering data races in candidate locations. It is important to emphasize the term "candidate", as *FastTrack's* hybrid approach may generate false positives, prioritizing completeness over preciseness.

FastTrack divides non-racing accesses to shared-variables into four categories — *write-exclusive*, *read-exclusive*, *write-shared* and *read-shared* —, which are then noised by *RaceInducer* after they occur. These locations are not noised proportionally, however, as *RaceInducer* effectively uses a heuristic to prevent the repeated exposure of the same data race, reducing the overall run time overhead. Its algorithm contains many other optimizations, with the aim of making it as efficient as possible, such as preventing ineffective noising by preventing noising during a long sequence of accesses to shared variable that have a low possibility of causing data races. Another optimization that effectively reduces the time threads spend waiting for locks, is only injecting noise into threads that are not currently holding any locks, with research showing that 97.1% of data races are associated with unprotected accesses [201].

3.3.5 CFLASH

CFLASH [72] is a tool developed by Garcia for testing concurrent Java software, developed by Rojas, as part of a master's dissertation. Its main purpose is to automatically detect and localize concurrency bugs, doing so by resorting to noise injection. As expected, the notion of "localizing" in this context refers to finding the actual interleavings that may lead to this error.

CFLASH was built based on the principle of the TNCS problem, which was briefly discussed in Section 3.1.2. This problem describes the challenge that exists when selecting the appropriate test cases and their parameters together with noise heuristics for a certain test objective [93]. In a nutshell, it reinforces the notion that no heuristic is a "silver bullet" that will be universally appropriate, and that programmers should strive to test as many heuristics, and parametrizations of these same heuristics, as possible, increasing the efficiency and efficacy of their testing process. This strategy is essential, if a programmer wishes to reach the maximum space state coverage possible, during their testing.

The CFLASH architecture is basically a pipeline, with processes outside and inside a Docker [146] environment. The usage of a dockerized environment is a sensible choice, as it allows for the abstraction of the tools inner-workings, portability and the lack of need to manage the tool's dependencies. Outside the dockerized environment, the user must choose a Java program, its test suite and a configuration file for CFLASH. Inside, CFLASH resorts to the static source instrumentation tool TXL, which was presented in Section 3.2.1.1, to add parametrizable noise instructions to the code. The program is then run multiple times, inside the dockerized environment, changing the noise parameters (i.e., seeding), and validating its output.

CFLASH supports the following noise placement locations: synchronized methods, blocks, statements accessing global variables and statements accessing objects passed as parameters. All of these placement heuristics are implemented through TXL's grammar. The only noise type it supports is *sleep*, although it allows for additional seeding capabilities, such as intensity and frequency (i.e., probability of triggering).

3.3.6 Ninja

Network noise INJection Agent (Ninja) [172] is a noise injection tool for detecting message races in concurrent software written resorting to the message-passing concurrent paradigm, which injects noise into the MPI (i.e., the network communication bus), in order to expose unintended message races. In a message-passing program, the trace of an execution is the ordered set of all the messages that were exchanged between processes [186]. Message races, like data races, can make a program unintentionally behave in a non-deterministic way. A message race occurs when the order of arrival at a process cannot be guaranteed and can be affected by such things as scheduling and network latency [154].

Message-passing concurrent software can run under Ninja without any previous modification and without being aware that the tool is being run. The authors put a considerable emphasis on this last point, due to the fact that usual tools introduce additional overhead into the program (node-local noise), resulting in noise that can effectively mask the data races it is trying to find.

There are two available noise placement heuristics in Ninja: system-centric and application-centric. The former emulates a congested network, while the latter analyses the application's communication pattern during a system-centric run, dumps the analysis data into a file and injects sufficient noise to cause the overlapping of each detected unsafe routine [171]. The application-centric is necessary, as the system-centric mode cannot ensure the overlapping of all pairs of unsafe routines. This is due to an observable phenomenon the authors refer to as the "separation problem" — the difficulty in reproducing unintended races when unsafe routines are "distanced" from each other, either due to the program's logic or added noise effects.

3.4 Comparison of Dynamic Noise Injection Tools

The following table gives a summarized high-level comparison of the different features of the dynamic noise injection tools previously described throughout Section 3.2.2. It is important to note, however, that the *sleep* primitive's intensity is implied by its duration, which is controllable. While for *yield* and *priority* primitives, the former has no parametrization and, therefore, no control. The latter does have a parametrization, but its outcome is non-deterministic and completely up to the scheduler.

Features	ConTest	ANaConDA	AtomRace	RaceInducer	CFLASH	Ninja
Programming Language	Java	C/C++	Java	Java	Java	C/C++
Paradigm	Shared-Memory	Shared-Memory	Shared-Memory	Shared-Memory	Shared-Memory	Message-Passing
Tool/Framework	Framework	Framework	Tool	Tool	Tool	Tool
Backend	—	Intel PIN	ConTest	RoadRunner	TXL	PMPI
Operating System	Cross Platform	Windows, Linux	Cross Platform	Cross Platform	Cross Platform	Linux
Current Status	Abandoned	Abandoned	Prototype	Prototype	Prototype	Abandoned
Licence	Proprietary	Open-Source	Unavailable	Unavailable	Open-Source	Open-Source
Noise Placement	Synch-block, Read/Write, Random	Read/Write, Function	Synch-block, Shared-vari, Random, Pattern	Shared-var	Synch-block, Read/Write	System-centric, Application-centric.
Noise Types	Sleep, Yield, Priority	Sleep, Yield	Sleep, Yield	Sleep, Wait	Sleep	Network Noise
Noise Frequency	Controllable	Controllable	Controllable	Controllable	Computed dynamically	Controllable
Noise Intensity	Controllable	Controllable	Controllable	Controllable	Computed dynamically	Computed dynamically

Table 3.1: High-level comparison of existing dynamic noise injection tools.

Chapter 4

Noise Injection For Java

Throughout Chapter 3, the dissertation presents noise injection as a helper method for analysing concurrent software and how it mitigates many of the issues that arise from relying on concurrent programming techniques. Additionally, the principles behind testing programs with noise injection tools are laid out. During this chapter, a study will be made of how these same principles can and should be applied, specifically when regarding the Java programming language. Understanding these principals will be quintessential when further applying them to a tool.

One of the main perks of choosing Java as the target programming language for a framework is the fact that it is a fully portable programming language, with existent JVM implementations for nearly every device in existence. This results in the tool developed being automatically cross-platform and capable of testing a language which runs on billions of devices worldwide. Java is a programming language with a heavy focus on concurrency, boasting a plethora of different synchronization constructs which may be used by a programmer when developing a concurrent application. From Garcia [72] and Goetz et al. [73], the following collection of the most notable Java concurrency constructs was gathered:

1. **Synchronized Blocks**
2. **Reentrant Locks**
3. **Synchronized Methods**
4. **Threads**
5. **Shared Variables**
6. Semaphores
7. Latches
8. Barriers
9. Exchangers
10. Synch. Collections
11. Futures
12. Concurrent Collections

By examining this list, it becomes immediately obvious that a considerable number of constructs exist, requiring this problem to be approached with a clear strategy on what types of noise primitives to inject and which heuristics to use when doing so. Given that the development of this project is bounded by a time constraint, it becomes infeasible to design heuristics for every single one of these primitives and, afterwards, implement them in a tool. As such, the choice was made to opt for following an incremental strategy, incrementally designing and implementing functionalities through short development cycles. There will also be a prioritization of some

concurrent constructs' implementation over others, as demonstrated by the enumeration present in the list above. In the end, noise placement strategies were designed and implemented for the top five constructs in the list.

The heuristics implementations for each of these constructs share very little programming logic between them, requiring unique and non-reusable code, which increases overall design and development time significantly. Additionally, not all heuristics can or make sense to be applied to every construct, requiring a case-by-case study on both their relevance and feasibility, regarding a particular construct.

4.1 Java Primitive Noising

This dissertation proposed a taxonomy for the noise placement component of heuristics in Section 3.1.2. While the taxonomy applies to any language, it is necessary to concretize it to the Java programming language, in order to accommodate for the specifies behind its design choices. For example, in a heuristic, "synchronization-based" is a noise placement category, and "after a synchronized method call" is a specific noise placement location.

During this section, several noise placement locations for noising programs are presented. Some of these heuristics are already established, being present in notable noise injection frameworks, while some are proposed as part of this dissertation's contributions.

The examples, instead of illustrating how a specific noise type would be inserted, use the `noise()` method abstraction. It is then up to a potential developer to choose if the noise seeding logic will be hardcoded either explicitly, or through an abstraction method which communicates with a logic module, responsible for dynamically controlling and triggering noise seeding.

Some code examples will also be shown, allowing the reader to gain some insight into how these instrumented statements would look like in "regular" source code, which is a program representation no doubt most are more familiar and, therefore, more comfortable with.

4.1.1 Synchronized Methods and Blocks

The `synchronized` keyword, inspired in the already decades old concept of monitor, is one of the main synchronization mechanisms Java offers to its developers, when constructing concurrent programs. In Java, every object, has an intrinsic lock associated with it. The `synchronized` keyword defines regions where a thread needs to acquire this intrinsic lock in order to continue.

In Java, the `synchronized` keyword can be used in two different scenarios: in a method's signature or as a block inside a method's body. The two variants shall be henceforth referred to as "synchronized methods" and "synchronized blocks", respectively.

Inserting noise before and after calls to synchronized methods or synchronized blocks increases the probability of triggering different interleavings, as the ordering of concurrent accesses to the same object monitor can see their order changed. Additionally, from these additional orderings, it becomes possible to expose latent bugs, stemming high-level data races. This risk becomes even more serious, given that these primitives are very commonly used. Both the ConTest [58]

and CFLASH [72] noise injection tools allow for the insertion of noise into these Java concurrency primitives.

4.1.1.1 Synchronized Methods

When the `synchronized` keyword is added to a method's signature, every time the method is run, it needs to acquire that object's monitor. If the method is static, it must acquire the monitor for the actual class and not a specific instance of an object.

Listing 4.1 illustrates how a synchronized method call could be noised in Java, by inserting noise before and after the method call to the `foo` synchronized method.

```
synchronized void foo() {}

void routine() {
    foo();
}
```

(a) Synchronized method call without noise

```
synchronized void foo() {}

void routine() {
    noise();
    foo();
    noise();
}
```

(b) Synchronized method call with noise

Listing 4.1: Noising synchronized method calls.

4.1.1.2 Synchronized Blocks

While the monitor for synchronized methods is implied by the object or class which declares it, the synchronized blocks allow for securing monitors for any object inline. It is important to note that these are not separate monitors, and a synchronized block and a synchronized method body, targetting the same object, cannot be run concurrently, as they will try to obtain the same monitor.

In Listing 4.2, an example of this mechanism is used to explain its syntax. By providing the `button` instance of the `Button` class as a parameter, it acts as a barrier, forcing threads to wait until they obtain the monitor for the `button` object, before proceeding to execute the routine.

```
Button button;

synchronized (button) {
    button.click();
}
```

Listing 4.2: An example of a synchronized block.

The technique for noising these blocks is rather similar to the synchronized method calls, with noise being inserted before and after reaching the synchronized block. Listing 4.3 demonstrates how noise would be inserted in such a block. In this specific case, the `this` keyword is used,

meaning that, upon reaching the block, the thread will attempt to attain the monitor which belongs to the instance of the object whose class declares the method where body this block is in.

```
void routine() {
    synchronized(this) {
        System.out.print();
    }
}
```

(a) Synchronized block without noise

```
void routine() {
    noise();
    synchronized(this) {
        System.out.print();
    }
    noise();
}
```

(b) Synchronized block with noise

Listing 4.3: Noising synchronized blocks.

4.1.2 Threads

In Java, threads are created via the `Thread` class and its respective interface. It is possible, however, not to access this primitive directly and instead use abstractions, such as an `Executor`, which provides a solution for a smart self-managing thread pool.

The `Thread` class in Java implements the `Runnable` interface, which defines a `run` method which contains the code that shall be executed when the thread is started via a `run` or `start` method invocation.

Given two threads T_a and T_b, where thread T_a is responsible for executing the statement that will launch T_b, the proposed thread-based noise placement startegy defines the following locations for Java Thread events:

- Inside T_a's routine, immediately before thread T_b's launch statement, with the purpose of delaying its launch;
- Inside T_a's routine, immediately after thread T_b's launch statement, with the purpose of delaying T_a's next statement (which may very well be the launch of additional threads);
- At the beginning of the code executed by T_b, with the purpose of delaying its routine but not T_a's.

While it is also possible to insert noise inside T_b's routine, right after its last statement, this would not be of much use, as there cannot be any synchronization errors since T_b is not executing any operation. It could, however, delay a thread's release and force the JVM to hold its resources for longer, saturating the system. But these types of analysis are out of the scope of this work.

Given the previously defined locations, Listing 4.4 will illustrate, with actual code examples, how the noise would be inserted into the three proposed locations.

Given the sheer amount of existing libraries which make use of Java threads, including the ones bundled with the JDK, which are part of the language (e.g., `Executors`), it would be unfeasible to showcase how the noising would be done for each of these abstractions. As such, it is expected

```
void routine() {
    Thread t = new Thread(new Foo());
     for (int i = 0; i < n; i++)  {
        t.start();
    }
}


class Foo {
    void run() {
        // ... Foo.run() routine
    }
}
```

(a) Thread call without noise

```
void routine() {
    Thread t = new Thread(new Foo());
    for (int i = 0; i < n; i++)  {
        noise();
        t.start();
        noise();
    }
}

class Foo {
    void run() {
        noise();
        // ... rest of Foo.run() routine
    }
}
```

(b) Thread call with noise

Listing 4.4: Noising the Java Thread primitive.

of a developer to dissect these thread-management libraries and understand where these events are occurring.

It is also worth mentioning that inserting noise before and after threads are launched can be very detrimental both to performance and the efficacy of interleaving generation. This may be evidenced in the routine of many concurrent programs relying on a thread pool, created *a priori*, to execute their workload. If these threads are launched in a loop, noising before and after threads are launched will mean that, not only one of the noise statements becomes redundant, but the program may become serialized.

4.1.3 Reentrant Locks

`Reentrant Locks` are a type of Java concurrency primitive which implements the `Lock` interface, with the intent of providing a more fine-grained control over lock management by adding more features and functionality than conventional solutions involving the `synchronized` keyword. The concept of "reentrance" describes the possibility of the same thread acquiring the same lock multiple times without realising it, avoiding situations in which a thread would otherwise block itself [155]. Other functionalities include the possibility of interrupting a thread that is waiting to acquire a lock and of implementing a lock acquisition fairness policy [73].

Since the `Reentrant Lock` Java primitive relies on a class to handle all of its functionality, noising this primitive is essentially a question of noising the relevant method calls. In this case, these are the calls to the `lock()` and `unlock()` methods, and the proposed noising locations to be prior to the `lock()` and following the `unlock()` method calls.

The strategy for noising the `Reentrant Lock` Java primitive is illustrated in Listing 4.5. Choosing to insert noise inside the `Reentrant Lock`'s block would be counter-productive, as it would just make its routine take longer to execute, severely affecting efficiency, without providing any advantage in terms of affecting the scheduler's ordering.

```
ReentrantLock lock = new ReentrantLock();

void routine() {
    lock.lock();
    baz();
    lock.unlock();
}
```

(a) Reentrant lock routine without noise

```
ReentrantLock lock = new ReentrantLock();

void routine() {
    noise();
    lock.lock();
    baz();
    lock.unlock();
    noise();
}
```

(b) Reentrant lock routine with noise

Listing 4.5: Noising reentrant locks.

4.1.4 Shared Variables

The noising of shared variables is a subtype of the Read/Write-based noise injection heuristic, previously covered in Section 3.1.2, targetting specifically read and write accesses to variables, which are either known to be shared or can be considered as very likely to be shared.

Research in the area of concurrency has generated a plethora of different methods for detecting shared variables. The ConTest [58] framework contains a method for noising shared variables.

Our proposed shared variable algorithm focuses on noising dependencies between variables by constructing a dependency graph from variable accesses. Given the added complexity of this method, the algorithm's logic was partitioned into three separate phases: dependency graph construction, dependency graph flattening and noising. These three phases will be thoroughly explained below.

4.1.4.1 Dependency Graph Construction

This first step consists in building a directed cyclic graph (DCG) representing all the dependencies between variables. The definition of a dependency, in the context of this heuristic, can be observed in Definition 2.

Definition 2 (Field dependency).

A program contains a field dependency iff:

$$\forall\, s \in S\ \exists\, s_L, s_R \in s :$$
$$\left(D_{s_L} \cap D_{s_R} \neq \emptyset\right) \vee \left(D_{s_L} \cap s_L \neq \emptyset \vee D_{s_R} \cap s_R \neq \emptyset\right)$$

- S - *set of all statements, which are either assignments or comparisons, in the program*
- D_x - *set of all the dependencies in a variable or set of variables* x
- s_L, s_R - *sets of variables referenced in the l-value or r-value of a statement* S

To find all the dependencies between variables, it is necessary to analyse every single assignment or comparison in the program. While the l-value of a Java program usually only references one variable, the r-value may contain multiple variables. Since not all r-values are in the form of

field references (e.g., they may be local variables or method invocations with multiple arguments), it is necessary to also keep track of the dependencies between these other forms of r-values. For this, the graph contains entries for locals and for method return values. It is thus necessary to extract every single variable in both parts of the statement. The algorithm for building the dependency graph is contained in Listing 4.1.

Algorithm 4.1 Dependency Graph Construction Algorithm

```
1:  M_p                                                  ▷ Set of methods for a program p
2:  S_m                                                  ▷ Set of statements for a method m
3:  DCG_p ← { }                                          ▷ Directed Cyclic Dependency Graph for a program p

4:  for all m ∈ M_p do                                   ▷ Traverse each method m
5:      for all s ∈ S_m do                               ▷ Traverse each statement s in method m's body
6:          if s.IsAssignment() then
7:              l ← S_L                                  ▷ Get l-value of s
8:              R_v ← GetReferenced(S_R);                ▷ Get all variables and method calls referenced in s's r-value
9:              for all v ∈ R_v do                       ▷ Add all referenced variables as dependencies
10:                 AddDependencies(l, v)
11:         else if s.IsReturnStatement() then           ▷ Get variables referenced in return statements)
12:             R_v ← GetReferenced(s);                  ▷ Get all variables and method calls referenced in return statement
13:             AddDependencies(s.GetParentMethodSignature(), v)
14:         else if s.IsConditional() then               ▷ Conditional statement (e.g., if or while)
15:             ...                                      ▷ This specific case is left for future work
16: function AddDependencies(l, v)
17:     if v.IsMethodInvocation() then
18:         s ← v.GetMethodSignature()
19:         for all p ∈ v.GetParameters() do
20:             AddDependencies(s, p)
21:         DCG_p[l] ← DCG_p[l] ∩ s
22:     else
23:         DCG_p[l] ← DCG_p[l] ∩ v
```

An example of a simple program, can be observed in Listing 4.6. The dependency graph for this program can be witnessed in Figure 4.1. Nodes `A`, `B` and `C` are relative to the declared fields, nodes `L1`, `L2`, `L3` and `L4` are locals, which are only witnessable in bytecode, and, finally, `R1` is a pseudo-type variable created for keeping track of method returns dependencies.

```
public static int A;
public static int B;
public static int C;

public static void main() {
    A = B;
    B = c();
    C = A;
}

public static int c() {
    return C;
}
```

Listing 4.6: Shared variable heuristic program example.

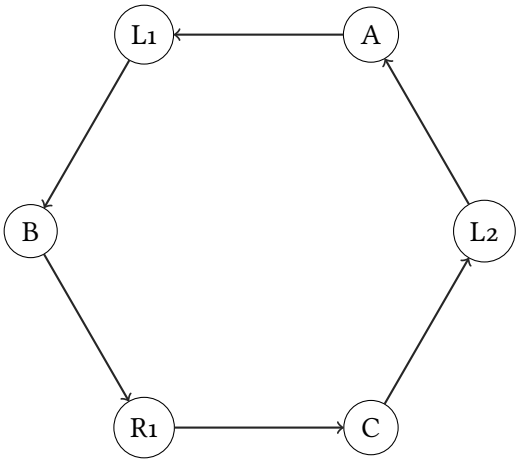


Figure 4.1: Shared variable heuristic program example's dependency graph.

Until now, this section only covered shared variables accesses stemming from assignments. However, not all accesses to shared variables are done through assignments. An example can be witnessed in Listing 4.7, where it becomes evident that, through the `if`'s condition, the value of the variable A depends not only in the value of C, but also of B. This whole `if` statement block should be, in order to guarantee correctness, inside a properly locked atomic region. The respective code example's dependency graph can also be visualized in Figure 4.2.

Inserting noise before blocks with conditional statements, such as this `if` statement, would allow threads to have a larger window to execute any pontentially conflicting statements, in between these two sequential accesses. Otherwise, since the accesses (conditional verification and subsequent assignment) are one right after the other, it is almost impossible to trigger conflicting interleavings. These are, as such, a source of latent concurrency errors.

```
public static int A;
public static int B;
public static int C;

public void routine() {
    if (A == B) {
        A = C;
    }
}
```

Listing 4.7: An example of a shared variable access through the condition of an if statement.

This logic also applies to any other type of conditional statement, such as `while`, `for` and `switches`.

4.1.4.2 Dependency Graph Flattening

Querying the DCG directly for variable dependencies when noising statements would be extremely inefficient, as it would be necessary to make a recursive search for all of a variable's

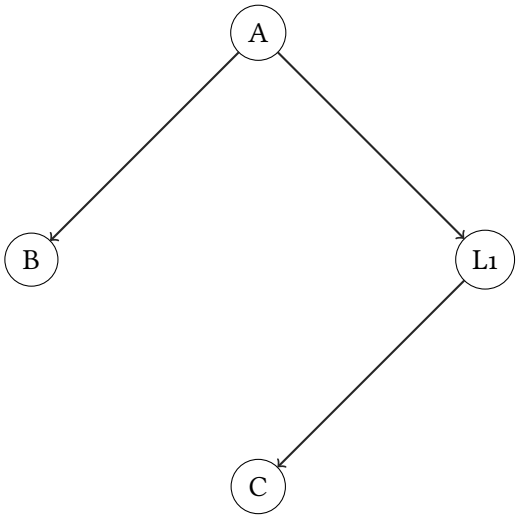


Figure 4.2: Shared variable heuristic example for conditionals.

dependencies. As such, it makes sense to "flatten" this data structure into another more efficient for querying (i.e., a map). In summary, the graph analysis step is responsible for parsing the complete dependency graph and mapping the aggregate of all of a variable's direct and indirect dependencies to the variable itself.

This requires exploring the graph recursively, avoiding loops such as the one present in Figure 4.1. Since the algorithm currently only concerns itself with variable dependencies, only these are added to each variable's dependencies in the map. This means that local and method returns dependencies are not used after the graph is fully constructed. The resultant dependencies from Listing 4.7 are the following:

$$D_A = D_B = D_C = D_{L1} = D_{L2} = D_{R1} = \{A, B, C\}$$

In this particular case, the dependency graph is circular and involves all declared variables, meaning every variable depends on every variable, including itself. Other programs may present more complex dependency graphs. Very large programs will result in an even larger dependency graph, since, as witnessable by the differing number of variables relative to nodes, the increase in number of nodes is not exactly linear, as many locals and method returns may be in the program logic.

The algorithm for this step is very straightforward and is basically a recursive search of the dependency graph, which maps the aggregate of all variable dependencies to the variable itself. The algorithm is demonstrated in Listing 4.2.

4.1.4.3 Shared Variable Noising

After the DCG with all the dependencies has been generated and parsed, a map containing every field, mapped to the set of its dependencies, is obtained. This phase goes through the body of every method and extracts all the assignments. Each assignment is then sequentially processed, and every variable contained in the expressions that make up both the l-values and r-values are

Algorithm 4.2 Dependency Graph Mapping Algorithm

```
1: DCG_p                                          ▷ Directed Cyclic Dependency Graph for a program p
2: DM_p ← { }                                     ▷ Dependency map for a program p

3: for all node ∈ DCG_p do                        ▷ Traverse each node of the DCG
4:     dependencies ← AddDependencies(node, dependencies)   ▷ Recursively search for all dependencies
5:     for all d ∈ dependencies do
6:         if d.isFieldOrLocal() && d ∉ DM_p[node] then     ▷ Filter for only field or local dependencies
7:             DM_p[node] ← DM_p[node] ∪ d
8: function AddDependencies(node, dependencies)
9:     for all edge ∈ node do
10:        if edge.Destiny() ∉ dependencies then
11:            dependencies ← AddDependencies(edge.Destiny(), dependencies)
12:            dependencies ← dependencies ∪ edge.Destiny()
       return dependencies
```

extracted. From these variables, which are the fields, locals and returns that were previously inspected to generate the graph, the aggregate of all their dependencies, generated in the last step, are extracted from the dependency map.

If the intersection of the sets of dependencies from the l-value and r-value is not empty, or one of the dependencies from the r-value or r-value depends on itself, it will mean that a variable dependency exists at this statement and, as such, this statement should be noised. For every statement that contains a common dependency between the l-values and r-values, noise should be inserted before and after the respective shared field access.

Listing 4.8 illustrates how the final noised program from Listing 4.6 would look like. Since the current algorithm does not perform an additional analysis step which checks which methods or code blocks are executed atomically or only by a single specific thread, this technique, while benefiting from its simplicity, falls short by possibly generating a lot of false positives. In this example, some noise statements are commented because, although they should be inserted there, there is already a statement right before. Noise statements should never be one right after the other, as this has the potential to insert more noise than intended, linearizing execution and lowering the efficacy and efficiency of noise injection.

4.1.4.4 Algorithm Complexity

It is important to note that the entire, three-part, algorithm necessary for noising shared variables, is relatively complex. As such, it would make sense to analyse both the temporal and spatial complexity of this algorithm. These are presented, for each relevant step, in Table 4.1. In this table, s represents the number of statements, v the number of referenced variables in a statement and n the number of nodes in the DCG.

As such, we may conclude that, in its current form, the algorithm has a quadratic temporal complexity and linear spatial complexity. Still, it may be possible to further improve this algorithm with more efficient logic or specialized data structures.

```
public static int A;
public static int B;
public static int C;

public static void main() {
    int L1, L2;

    noise();
    L1 = B;
    noise();

    noise(); // redundant
    A = L1;
    noise();

    noise(); // redundant
    B = c();
    noise();

    noise(); // redundant
    L2 = A;
    noise();

    noise(); // redundant
    C = L2;
    noise();
}
```

Listing 4.8: Shared variable heuristic program example with noise.

Table 4.1: Shared variable noising algorithm's spatial and temporal complexity.

Algorithm Step	Temporal Complexity	Spatial Complexity
Parsing every relevant statement	$O(s)$	-
Building the DCG	$O(v)$	$O(2v-1)$
Parsing the DCG to construct a Map	$O(n^2)$	$O(n^2)$
Parsing every relevant statement again	$O(s)$	-
Checking statement dependencies	$O(v)$	-

4.2 Program Trace Analysis

In a noise injection framework, a program tracing mechanism is almost as important as the noising mechanism itself. If there is no implemented method for tracking how much of the state space is actually being covered, it becomes impossible to ascertain how effective a noise-injection-based testing technique actually is. Additionally, tracing allows the tester to check if a given heuristic is more or less effective or efficient than another. It is not only important to create a valid tracing approach but also to create a valid trace interpretation approach, which can simplify the trace in order to reason about understanding it [96].

There are two main important uses for a trace, the first being comparison between runs and coverage analysis and the second being reproducibility, which allows a tester to replay a given

interleaving. This last step is vital for a tester, as it provides a tester with the assurance that, upon finding a latent error (triggered in a small state space partition), he may replay that given run or runs and verify that the bug fix was effective.

This section tackles the tracing problem by separating it into three phases: trace construction, trace interpretation and manipulation and, lastly, trace comparison. The trace replay mechanisms are out of the scope of this work and left for future work.

4.2.1 Trace Construction

The most straightforward approach to constructing a program trace, would be to construct this trace with the context information upon the program arriving at regions where context switching is most likely to occur. But, since the program already inserts noise at locations which are the most likely to influence the scheduler (i.e., synchronization locations), the most straightforward and logical approach would be to create a trace from the locations where noise is inserted.

From the program's constructed and outputted trace, it is then possible to infer how different a given run is from another, in terms of event ordering. A solid minimalistic approach would define an event as a specific thread passing through a tracing location. From a list of (thread ID, noising location ID) pairs, it is then possible to extract the given run's thread interleaving.

Let us assume a solution where the trace is composed by the (thread ID, noising location ID) pairs. A necessary requisite of these IDs, which compose the pair, is that they have to be unique. If the IDs are not unique, any attempt to compare them will almost certainly result in collisions. From this description, a program trace might look like this, where t_x is a thread with ID x:

$$\{(t_{12}, a), (t_1, a), (t_3, a), (t_2, b), (t_15, c), (t_{15}, a)\}$$

For each pair, the first value represents the ID of a thread, while the second represents the ID of a given noising region. For simplicity, thread IDs are numeric, while noising location IDs are alphabetical. The length of the IDs is irrelevant at this phase and will be tackled in the next section.

4.2.2 Trace Interpretation and Manipulation

The previous step left us with a close to usable program trace. However, with a few additional modifications to the generated trace, it will then become "comparison friendly".

When the JVM assigns an ID to a thread, it guarantees ID uniqueness but is non-deterministic, since multiple runs may result in different IDs assigned to threads performing the same routines. The trace example from the last section, was tailored to illustrate this behaviour, as the first thread that creates a trace entry, is numbered with the ID "12". A simple way to deal this issue, is to map these IDs to new and unique incremental IDs, based on the order of their appearance. In [115], the authors also devise a strategy to identify threads, since between various runs, the JVM-assigned thread IDs will be non-deterministic. However, their strategy is more geared toward identifying threads based on their execution trace and less so on a method for ordering them.

An additional bonus of this method is that some equivalent interleavings are eliminated, shortening the state space very significantly. Although there are many proposed definitions, both syntactic and semantic, for what are equivalent interleavings, ours is syntactic and quite simple. We define equivalent interleavings as those where the event sequence is the same but the corresponding thread IDs may differ. This is a very common occurrence in Java, given the JVM's non-determinism when assigning these thread IDs. An example of two equivalent interleavings, represented by sets of (thread ID, noising location ID) tuples, with two threads t_1 and t_2 executing the same code containing two instrumentation points a, b and c, is $\{(t_1, a), (t_1, b), (t_2, a), (t_2, c)\}$ and $\{(t_2, a), (t_2, b), (t_1, a), (t_1, c)\}$.

The last modification, transforms a pair of two unique IDs into just one unique ID identifying that pair, removing unnecessary redundancy. The total length of the trace will lower significantly and comparison results will be more accurate if these (thread ID, noising location ID) tuples are mapped to unique identifiers.

This step will ultimately transform the previous sequence:

$$\{(t_{12}, a), (t_1, a), (t_3, a), (t_2, b), (t_1, c), (t_{12}, a)\}$$

Into the following sequence:

$$\{A, B, C, D, E, A\}$$

After this mapping process is completed, a program trace will be a string composed of a set of unique characters, which will provide a much more precise insight into how different interleavings are, when a string comparison algorithm is used to calculate either their distance or likeness ratio.

4.2.3 Trace Comparison with String Comparison Metrics

String comparison metrics are found in use by the industry in a multitude of scenarios, from fraud-detection to text auto-correction, with most modern spell-checking solutions relying on some kind of string distance metric to compare word distance [50]. As such, several of these string comparison algorithms have been proposed so far. During the remainder of this section, some of the most notable examples will be presented and explained.

This section is divided into two parts. The first part presents existing notable string comparison metrics, contextualizes them in the intended scenario of comparing traces in order to calculate trace likeness. After analysing notable string comparison metrics and deducing the most adequate for trace comparison tests, the method of comparison is explained in the second part of this section.

4.2.3.1 String Comparison Metrics

Hamming Distance The Hamming Distance [77] metric is the most straightforward of this set. The Hamming distance between two strings is the amount of indexes, in both string, whose respective characters differ, similarly to a bitwise AND operation. Although the Hamming distance algorithm technically requires two strings with the same length, the algorithm can be extended to pad the missing spaces with empty characters. Some examples follow:

- ham(ABC, ACB) = 2, since two characters differ;
- ham(AAA, BBB) = 3, since all characters differ;

In our intended scenario, this metric suffers from its main perk — simplicity. Traces do not necessarily always have the same number of operations, as conditional statements can cause a program to have unexpected different flows. By design, Hamming distance does not work well when insertions and deletions are required for string matching, being more suited for scenarios where most of the strings sequences are expected to be aligned [86]. This effectively makes Hamming distance the least ideal method for calculating the ration of likeness between two traces.

Levenshtein Distance The Levenshtein distance [129] is one of the most famous string comparison metrics. The Levenshtein distance between two strings is the minimum amount of single-character edits (i.e., insertions, deletions and substitutions) necessary to transform the first string into the second. The upper bound of edits is the length of the longer string, while the lower bound is the difference of the length between the two strings. A few examples follow:

- lev(AAA, ABA) = 1, since only one letter has to be substituted;
- lev(AAA, AA) = 1, since a letter has to be removed;
- lev(ABC, ACB) = 2, since two letters have to be substituted.
- lev(AA, ABA) = 1, since only one letter has to be inserted.

The last example highlights a very big perk of the Levenshtein distance when compared to the Hamming distance, which is the possibility of inserting/removing characters in the middle of a string. This remains true even when the latter allows for the padding of empty characters. However, this method is still not ideal, when we realize that the Levenshtein distance between $\{ABC\}$ and $\{ACB\}$, is the same as the distance between $\{ABC\}$ and $\{ADF\}$. Such small differences in trace location ordering should not impact the distance as much.

Damerau–Levenshtein Distance The Damerau–Levenshtein [48] distance is a notable variation of the Levenshtein distance, where the algorithm is extended to allow an additional operation: character transposition. These transpositions are, however, limited, in range, to adjacent characters. The following examples illustrate this behaviour:

- dam_lev(ABC, ACB) = 1, since the two letters are adjacent, meaning they can be transposed.
- dam_lev(AAB, BAA) = 2, since the two letters are too far away to be transposed, thus requiring two substitution operations.

The added functionality of this method is highly desirable for interleaving comparison purposes, as it can yield a significantly lower distance value for two very similar traces.

Jaro Similarity The Jaro [97] similarity, is a ratio of how similar two strings are, represented by a floating point value v, with $0 \leq v \leq 1$, where $v = 1$ means the two strings are equal.

The formula for the Jaro Similarity is relatively more complex, taking into account the matching characters in both strings, their respective ordering and the difference in length. The formula is the following:

Definition 3 (Jaro similarity, from [97]).

$$J = \begin{cases} 0 & for\ m = 0 \\ \frac{1}{3}\left(\frac{m}{L_{S_1}} + \frac{m}{L_{S_2}} + \frac{m - \frac{t}{2}}{m}\right) & for\ m \neq 0 \end{cases}$$

- S_n - *string to be compared, with* $n \in 1, 2$
- L_{S_n} - *length of string* S_n
- m - *number of matching characters in* S_1 *and* S_2
- t - *number of transpositions necessary to transform* S_1*'s matching character sequence into* S_2*'s*

From the formula, it becomes clear that the Jaro similarity's algorithm favours strings which have matching character sequences. Additionally, Jaro's algorithm allows for transpositions of characters farther apart. The Jaro similarity is also interesting because it does not penalize as much the existence of a differing order of events (e.g., conditional statements or loops) inside otherwise equal sequences, even if the final strings end up with very different lengths. Some examples follow:

- jaro($ABCCCDE$, $ABCDE$) $\approx$ 0.9, simulating a small loop.
- jaro($ABXYZDE$, $ABCDE$) $\approx$ 0.79, simulating differing logic.
- jaro($ABCCCCCCDE$, $ABCDE$) $\approx$ 0.63, simulating a long loop.

Of all the researched string comparison metrics, Jaro similarity presented itself as the best candidate for calculating interleaving likeness. While it shares some drawbacks with the Levenshtein distance, such as the distance between the strings ABC and ACB being equal to the distance of the strings ABC and ADF, it handles loops in a much more desirable way. Additionally, in [40], Cohen et al. demonstrated that the Jaro string metric outperformed the Levenshtein string metric, in a scenario where names and records were matched.

Jaro-Winkler Similarity The Jaro-Wrinkler [197] similarity is an adapted version of the Jaro algorithm. This new algorithm contains two new parameters — δ and ρ —, which represent the matching prefix length and scaling factor, respectively. As such, by tuning the scaling factor ρ, it becomes possible to give a more favourable rating to strings which have a longer matching prefix. A few examples follow:

- jaro_wr($ABCDE, ABCDX$) $\approx$ 0.92, with differing logic at the end.
- jaro_wr($ABCDE, XBCDE$) $\approx$ 0.87, with differing logic at the start.

The use of this variant for calculating trace likeness will mean that the inadvertent assumption will be made, that a matching prefix in two traces is more significant to their likeness than a similarly sized affix at some other location. This is not a desirable feature, when comparing how similar traces are.

4.2.3.2 Trace Comparison

After a trace is simplified, with its (Thread ID, trace location ID) pairs converted to single characters, it is possible to represent them as a string value. This string value is a sequence of unique characters, which can be fed to a string comparison algorithm. These string comparison algorithms are the heuristic which will determine the string distance between two trace representations, allowing us to deduce interleaving likeness. It is possible to extract two main metrics from the trace information:

Number of Unique Interleavings This metric represents the number of x non-identical interleavings, in y runs, where $x \leq y$. A higher number of unique interleavings directly correlates with a higher amount of the program's state space being explored. However, it is important to notice that interleavings are defined only by the trace locations captured, meaning that equivalent interleavings using this metric may actually be different if we consider the full set of program states;

Average Interleaving Distance This metric measures the distinctness of the observed interleavings resorting to a string comparison algorithm, from which the Jaro [97] metric was deemed the most effective of all researched metrics.

This distance j, with $0 \leq j \leq 1$, is lower proportionately to how distinct two strings are from each other. If we were to compare all traces with each other (an algorithm with quadratic temporal complexity), and extract the average distance, it would be possible to understand if an heuristic generates more or less similar interleavings than another.

Given a series of program traces X, with size n, and a distance metric D, the formula for calculating the average interleaving distance is as follows:

Definition 4 (Average Interleaving Distance).

$$AvgDist(X_1, ..., X_n) = \frac{\sum_{i=0}^{n} \sum_{j=i+1}^{n} D(X_i, X_j)}{n^{\frac{n}{2}}}$$

4.2.3.3 Notes on Trace Comparison

While a string comparison metric can compare two traces and extract a value representing a level of similarity, this level is the product of merely syntactic comparison. This essentially means that no importance is given to the probability of an interleaving to occur (i.e., its "rarity") or how distant the trace points may be in code and/or time.

A trace distance is always bounded between a maximum and minimum value, which will depend on the algorithm used for calculating the trace difference. As such, we can expect that a complete exploration of the state space of a program should yield an average distance which will tend to half of the traces' length. Unfortunately, so too will an exploration which only finds two traces, with one being being entirely different from another. While its possible to identify some of these scenarios by observing the standard deviation, this means that the average interleaving distance metric is rather meaningless by itself.

One could also naïvely assume a higher average interleaving distance between a series of traces is always better. However, this does not hold true at all. A higher average interleaving distance informs us not only that we are triggering interleavings which are more similar to each other, but also that we are not triggering interleavings which are less similar to each other. Ideally, a heuristic should be tuned for triggering interleavings which are both more and less similar to each other, showing a tendency to, given enough runs, managing to explore the entirety of the program's state space.

In conclusion, the average interleaving distance metric can only be used as a measure of state space exploration when paired with other metrics. It is impossible to guess if in 1000 runs yielding 2 unique interleavings, each occurred 500 times or one was triggered 999 times and the other just 1. As such, it is necessary to evaluate the average interleaving distance together with the standard deviation and the number of unique interleavings observed.

In specific instances, some programs may exhibit traces which have parts which are serialized. Either the tracing is being activated at a noising/tracing location which shouldn't be active, or there is a deterministic routine, such as launching a series of threads. Nevertheless, in these scenarios, a mechanism should be employed which detects long common subsequences at either start or end of traces, and removed them from the comparison, as these may influence the similarity level in some metrics (e.g., Jaro).

Chapter 5

OSCAR — A Java Noise Injection Framework

During the last few sections of Chapter 3, some of the most notable noise injection tools and frameworks were presented and afterwards summarized in Table 3.1. Upon closer examination of this table, it becomes immediately noticeable that, to the extent of the author's research, the only noise injection framework for the Java programming language consists of *ConTest* [58]. Given the fact that *ConTest* is a proprietary and abandoned concurrency testing framework that is, as of now and to the extent of the author's knowledge, completely unavailable to the public, it is then possible to infer that there is a clear lack of availability of noise injection frameworks for concurrent software written using the Java programming language. This is very unfortunate, as frameworks have the potential to allow other researchers to build their own solutions on top, similarly to how *RaceInducer* [68] was built on top of *ConTest*.

As a means of providing a solution to the aforementioned problem, this dissertation proposes OSCAR [52] — a completely new open-source shared-memory-oriented static analysis framework for testing concurrent Java programs through noise injection, relying on bytecode instrumentation. Through the implementation of OSCAR, it is possible to apply all the insights which were discussed in Chapter 4. Therefore, as expected, the main drive for undertaking this project was to further research in the field of noise injection in Java, investigating proposed noise injection heuristics, understanding how to apply them to Java concurrency primitives, proposing novel variations of these heuristics, and researching methods to evaluate their impact. To do so, however, we were faced with several challenges, including: how to design a generalized instrumentation solution; which methods/statements/primitives to instrument; where noise should be localized; how to handle edge-cases in instrumentation; what are the pitfalls of noise injection; and how to measure and evaluate the impact of noising, e.g., coverage of the interleaving space.

The main objective of this dissertation is to create a framework that can not only be used for stand-alone testing, but also as a building block for other, more opinionated, testing tools. We envisage OSCAR as being both a tool to empower researchers in the field of concurrency and as a means for teaching concurrency at higher education institutions, sharing this motivation

with tools such as PIN [137]. OSCAR will be able to raise students' awareness to concurrency and the hazards of (improper) synchronization, by providing a medium for hands-on experience with noise injection and learning about the impact of non-determinism in masking errors in concurrent applications. Finally, a decision was reached to release OSCAR as a fully open-source piece of software, ensuring its unrestricted availability, while befitting from the application's continuous evolution through their feedback and contributions. The tool is available as a public git repository in the GitHub platform in the following URL: https://github.com/filipedeluna/oscar.

5.1 Soot's Architecture

It was established that the OSCAR noise injection framework should resort to static instrumentation, as dynamic instrumentation would require custom JVM implementations. Next, there was a choice of having OSCAR work at a bytecode level in order to allow for a more fine-grained level of access when injecting noise into Java's built-in concurrency constructs. The main reasons for choosing bytecode over source instrumentation, were the following: ease of heuristic development, as language semantics are simplified, when compared to the original code, where many language features are just "syntactic sugar"; compatibility with legacy code and libraries, since high-level language features do not always translate to bytecode differences; compatibility with other languages and frameworks which target the JVM, such as Scala, Kotlin, Groovy or Clojure; and the possibility to test any Java program, proprietary or not, without needing access to the original source code, as mentioned by Binder et al. in [17], where the authors present the advantages of resorting to bytecode instrumentation in order to add aspect-orientedness to programs.

The *Soot* bytecode instrumentation framework [188] was ultimately chosen to take on the role of OSCAR's instrumentation engine. Although Section 3.2.1.2, already contained a brief explanation of the *Soot* framework, it was from a very high-level perspective, leaving out important details and design choices of *Soot*, which will have a direct impact on any tool built up on top of it. Amongst *Soot*'s perks, when compared to other existing tools, one can highlight its extensive documentation, active development and community surrounding it. Since *Soot* currently supports Java 9 in bleeding-edge releases, the initial iterations of OSCAR will initially target this Java version.

Throughout this section, the reader will find a more in-depth description of *Soot*'s architecture and, more importantly, how OSCAR, as a solution, was built upon it.

5.1.1 Jimple

Jimple is currently the principal intermediate representation format in *Soot*, and the format which will be adopted during the development of OSCAR. Although *Soot* allows for the direct instrumentation of bytecode, resorting to an intermediate format, such as Jimple, alleviates much of the difficulty that stems from any necessary code analysis. Bytecode is naively translated to untyped

Jimple, whose syntax consists of less than a tenth of the number of statements present in bytecode [3]. Listing 5.1 shows a simple "Hello World" Java program that was compiled into a `.class` file and then transpiled, by *Soot*, into Jimple code. From this little piece of code, it is possible to observe how human-readable Jimple's syntax is, while simultaneous allowing a programmer the same degree of fine-grained control over code.

```
java.io.PrintStream $r0;
java.lang.String[] r1;
r1 := @parameter0: java.lang.String[];
$r0 = <java.lang.System: java.io.PrintStream out>;
virtualinvoke $r0.<java.io.PrintStream: void println(java.lang.String)>("Hello World");
return;
```

Listing 5.1: Hello World Jimple example.

Jimple is more than just an alternative syntax built with just presentation in mind. *Soot*'s internal library comes with an accessible API which provides a convenient object-oriented approach to code analysis. Each element of the syntax is represented as an object, with its specific methods and data-structures, allowing for hassle-free bytecode traversal. This enables the programmer with the tools necessary to implement code-search patterns programmatically, a feature which will prove vital when attempting to implement the instrumentation necessary for injecting noise into programs' code.

The main constructs for representing the bytecode syntax, implemented by the *Soot* architecture, are the following:

Units These are the program statements, which in Jimple are represented by the interface `Stmt`, and shared with the Dava and Grimple intermediate representations. For Jimple, dedicated classes implementing this interface were created, including, for example, invocation (`JInvokeStmt}`), conditional (`JIfStmt`), assignment (`JAssignStmt`) or return statements (`JRetStmt`);

Values These represent pieces of data such as local variables (`JimpleLocal`), constants (e.g., `LongConstant` and `NullConstant`) (`JimpleConstant`) and expressions (e.g., `JAddExpr` and `JAndExpr`). Since object variables in Java are references to objects in memory, these must also be handled as such, resorting to the `RefType` class in the *Soot* API. Likewise, when calling a method, one must also create a reference to it, to be passed on to the invocation expression. Jimple forces values to contain, at most, a single expression, facilitating analysis at the cost of readability [179];

References In *Soot*, these are called "boxes" and are responsible for holding collections of references to statements and values in *Soot*. Two different kinds exist: one for holding units (`UnitBox`), and another one for holding values (`ValueBox`). Every class contains a set of all

units and values that are used in it. Unit boxes are also used, for example, for branching or *goto* statements, when a reference to a target unit is necessary;

Traps These are used in order to handle Java exceptions, containing a reference to the thrown Exception object and handler.

Using the Jimple-specific implementation classes, comes with the perk of allowing for additional control, in the form of dedicated methods and properties, which can be used to manipulate the Jimple intermediate code.

5.1.2 Soot Phases

Patrick Lam, one of the designers of *Soot* [188], published a paper [158] which explained the design and architecture of the code transformation engine. This architecture is essentially a pipeline implementation, containing a sequence of different instrumentation phases, into which a programmer may inject additional instrumentation logic. In Figure 5.1, it is possible to visualize the main phases of the Soot instrumentation pipeline. Although other phases exist, such as the Grimp body creation and optimization, these require the programmer to set specific parametrizations. Also omitted from the diagram, for simplicity's sake, are all the sub-phases, that many of these phases include. The developers call these phases "packs", as each of them effectively pack a collection of code instrumentation logic, called "transformers"or "phases". Henceforth, we shall refer to them as packs and transformers, to avoid confusion.

While Soot never starts a phase before the previous' transformer routines have finished, it parallelizes regular transformers' routines, executing serially for every given method's body, traversing all of the program's original and dynamically created classes, behaving much like a parallel map. The exception to this rule, are the "whole-program" packs, whose transformers are executed serially and are meant for the instrumentation of the whole program, instead of a particular method body (i.e., analysing a method body vs. analysing a "scene"). It is worth noting that "Whole-Jimple" packs are only active if *Soot*'s engine is configured to run in whole-program mode, where the analysed program's libraries and frameworks may also be analysed [199]. OSCAR makes use of both of these types of packs, as parallelism is not always a desireable feature, since some transformers' may have conflicting routines, leading to data races. Unfortunately, the lack of parallelization does result in a considerable performance penalty, meaning "Whole-Jimple" packs should only be used for absolutely non-parallizeable workloads. One such example of a non-parallelizable workload, requiring whole-program analysis, is code instrumentation through recursive analysis of the program's callgraph.

Figure 5.1 shows a simplified version of the *Soot* pipeline. The packs which are signalled in red (the jimple transformation phase (jtp) and whole-jimple transformation phase (wjtp)) are the instrumentation phases into which OSCAR will inject additional and/or adapt existing code, allowing for its noise-injection. Every single noising type is achieved by implementing one transformer. The jtp is empty by default and, like the wjtp, has the benefit of allowing the developer to instrument the program while making use of the simplified Jimple syntax and

corresponding data structures. As stated earlier, the choice between resorting to the jtp or wjtp, will depend on whether the transformer's routine is parallelizable or not.

Tagging, is a feature made available by *Soot*'s API, which permits the developer to add (i.e. tag) values such as a string or number to one of the "Jimplified" Abstract Syntax Tree (AST)'s nodes. These tags allow for checks to be made at a later instance, at a method or statement level. This technique is used in some transformer implementations which will be discussed further along this document.

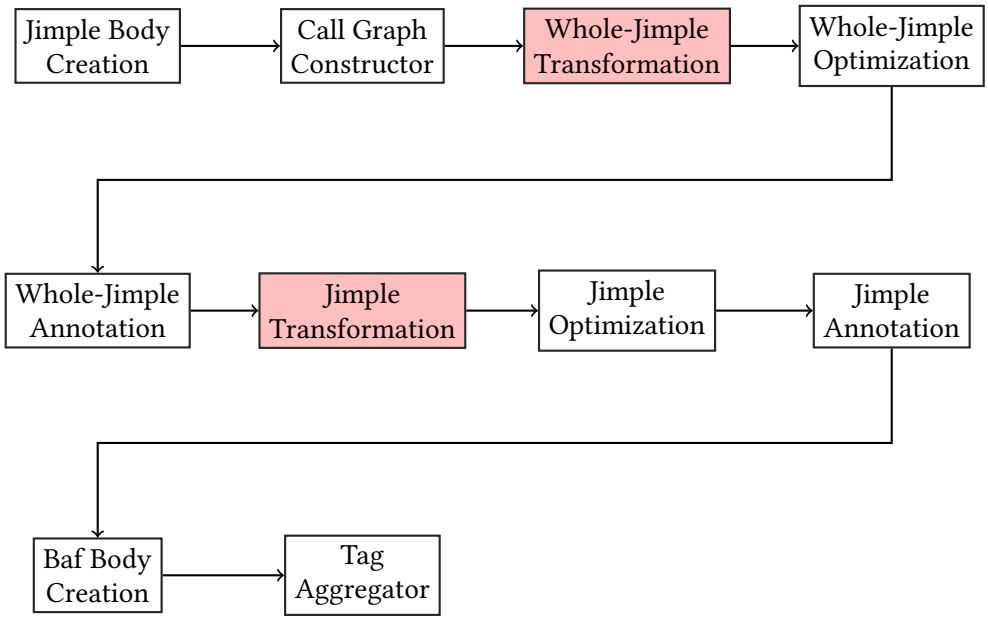


Figure 5.1: *Soot* Framework instrumentation phases (Adapted from [158]).

5.2 OSCAR

OSCAR's prototype was developed entirely from the scratch throughout the course of this dissertation, through an agile-like approach, resorting to small incremental updates. With every update, OSCAR was either refined, corrected or extended with new capabilities. This development strategy made perfect sense for OSCAR, as it provided the means to make short, weekly prototypes, which were then scrutinized by the whole team. However, to support this development approach, which demands parallel development (i.e, branching), for different features, OSCAR required a scalable architecture from the start. From this scalable architecture, features could be easily added, made better or removed, without jeopardizing the entire codebase.

5.2.1 Architecture

Given OSCAR's agile-like development approach, there was a need to greatly modularize its implementation. It was not at all feasible, given the available span of time, to cautiously verify the whole program every single update. The modularization of OSCAR's architecture was possible

in no small part thanks to *Soot*'s own architecture, given that its transformation pipeline allows for and encourages a modular approach to the development of program analysis tools.

An approach of instrumenting into the program every single statement of the overall noising logic, will quickly prove itself unsustainable. Since the bulk of this logic is redundant, it should be encapsulated in methods. Overall, the number of instrumented statements (i.e., instructions) injected into a program should always be kept to a minimum, as they are very hard to debug and can become a source of errors when added indiscriminately. As such, the bulk of re-usable injected programming logic should be inside a thoroughly unit-tested previously written "control" class, from which calls to its methods are then instrumented into the program. Together with the added benefit of greatly easing the challenge of verifying the injected logic, this approach also comes with the additional perk of making the programmatical instrumentation much easier.

Figure 5.2, contains a diagram illustrating the architecture of OSCAR's instrumentation engine. OSCAR takes a (compiled) Java Bytecode program as input, and runs a series of instrumentation steps, resulting in a new transformed/noised version of the same program. The following sections describe the implementation of each of the components described in Figure 5.2.

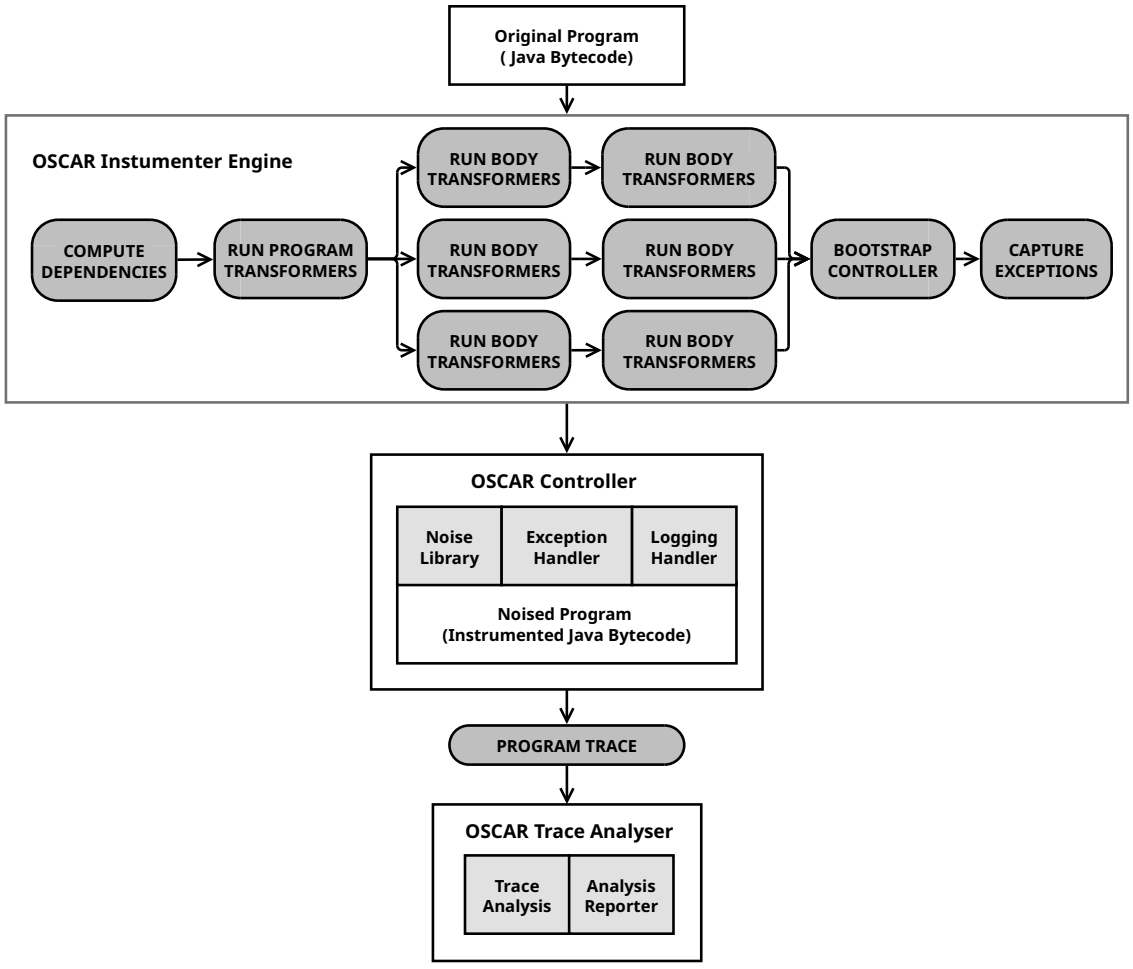


Figure 5.2: OSCAR framework architecture.

OSCAR can thus be broken down into three "main" components: a static instrumentation

engine (**Instrumenter**), a bootstrapped run time noise injector (**Controller**) and a trace analysis script (**Analyser**), which will be further elaborated later on in the chapter.

5.2.1.1 OSCAR Controller

Through the instrumentation engine of OSCAR's Instrumenter, a new instrumented program is created, containing the Controller bootstrapped to the original program. The Controller is OSCAR's dynamic noising engine which allows for the control of noise type, noise intensity, active noise placement locations and categories, output methods and other configurable parameters, when running the new noised program. The Controller logic can be broken down into the following components:

Noised Program: The original program's main method is replaced with a wrapper which passes arguments to the bootstrapped Controller. The OSCAR Controller can parse arguments passed into the program to parameterise its noise injection routine. Every instrumentation statement is implemented as a call to a Controller method, which contains the necessary logic. This approach allows for reusing and more easily maintaining the code responsible for the analysis of the noised program. Listing 5.2 illustrates how the Controller bootstrapping, done by OSCAR Instrumenter, modifies an example program.

```
public static void main(args) {
    System.out.println(args);
}
```

(a) Before OSCAR's instrumentation

```
public static void main(args) {
    try {
        main_wrapped(Controller.start(args));
    } catch (Exception e) {
        Controller.exception(e);
    }
    Controller.end(e);
}

public static void main_wrapped(args) {
    System.out.println(args);
}
```

(b) After OSCAR's instrumentation

Listing 5.2: OSCAR Controller bootstrapping.

Noising Library: Every single noise statement is implemented as a call to the `noise` method of this Controller. Noise instructions are statically instrumented into the program's bytecode, while the noise itself is triggered dynamically, as the program runs. The amount of noise triggered upon reaching a noising location is randomized, with configurable bounds.

The `noise` method takes as a parameter an hardcoded Universal Unique Identifier (UUID), uniquely representing it, along with the type of noise placement location, so that they can be enabled or disabled on a per case basis. This UUID is then used for tracing purposes, with the full program trace describing the execution order of these noising statements.

Tracing/Logging: Since the `noise` statements are already inserted at heuristically determined key synchronisation locations, the tracing piggybacks on the noising statement and can be configured to be triggered before and/or after the noising logic.

The trace information that OSCAR outputs, consists of a sequence of strings which represent the order of arrival to trace locations. These strings are each a tuple constituted by the JVM-assigned thread IDs and the trace location UUID.

Exception Handler: Since the program can crash or exit unexpectedly during its execution, all calls to `System.exit` are captured and the program's main method is wrapped in a try/catch block that captures every exception type. This mechanism allows for a graceful exiting, which is invaluable for a user to understand the reason for this abrupt exit. Such understanding can be derived from the interleaving which is represented in the trace, which is composed of information originating from calls to the Controller's `noise` method.

The second mechanism, relies on surrounding the method call to the instrumented program's `main` method by a try/catch block which is set to catch any given exception. After catching the exception, its graceful exit routine consists of outputting the trace of the exception, handling its internal finishing routine, like the first method, and finally exiting with an error signal.

```
static Counter counter = new Counter();

public static void main(args) {
    for (int i = 0; i < args[0]; i++) {
        Thread t = new Thread(Main::add);
        t.start();
    }
}

public static void add() {
    synchronized (counter) {
        Counter.add();
    }
}
```

(a) Before OSCAR's instrumentation

```
static Counter counter = new Counter();

public static void main(args) {
    try {
        main_wrapped(Controller.start(args));
    } catch (Exception e) {
        Controller.exception(e);
    }
    Controller.end(e);
}

public static void main_wrapped(args) {
    for (int i = 0; i < args[0]; i++) {
        Thread t = new Thread(Main::add_bootstrap);
        Controller.noise(BEFORE_THREAD_START, UUID);
        t.start();
        Controller.noise(AFTER_THREAD_START, UUID);
    }
}

public static void add_bootstrap() {
    Controller.noise(BEFORE_THREAD_ROUTINE, UUID);
    add();
}

public static void add() {
    Controller.noise(BEFORE_SYNC_BLOCK, UUID);
    synchronized (counter) {
        counter.add();
    }
    Controller.noise(AFTER_SYNC_BLOCK, UUID);
}
```

(b) After OSCAR's instrumentation

Listing 5.3: OSCAR noise instrumentation example.

Listing 5.3 depicts a simple program which launches a series of threads and increments their value inside a *synchronized* block, before and after being processed by OSCAR's instrumentation engine.

The final instrumented executable contains noise statements in before and after the thread launch, before the thread's routine and before and after the *synchronized* block. Additionally, it is possible to see how OSCAR wraps the original main method in order to provide a graceful exit mechanism.

5.3 Java Primitive Noising

The process of inserting noise statements and other logic which supports the noising process, is handled by the OSCAR Instrumenter. Every single inserted noising statement is a call to the Controller's `noise` method, which receives as a parameter the noise placement type and a UUID associated with the location in the code, hardcoded in the corresponding class file's bytecode. The routine then checks if the noise placement type is enabled and, if so, triggers with a randomized intensity, whose maximum and minimum bounds are configurable via parametrization. The program, when running, will issue every single call to the Controller's `noise` method, whether the noising location and/or heuristic is activated or not. It is then part of the Controller's routine to verify if the corresponding noise location and/or heuristic are active and, then, triggering noise if so. OSCAR supports tracing the program before and/or after the noise is triggered, in the Controller's `noise` method. As explained in Section 4.2.1, only active noise locations are traced, which is sensible, as these will then coincide with the synchronization locations most likely to be affected by the scheduler manipulation. The CPU time taken by the controller to perform these checks, when compared to the calls made to the thread scheduler, such as `yield()` and `sleep()`, is much lower, making its overall impact negligible.

According to Edelstein et al. [58], the *sleep* primitive has been shown to be the most effective at triggering additional interleavings, when executing an instrumented program multiple times. However, this efficacy comes at the cost of a significant slowdown of the execution [65]. As such, besides *sleep*-based heuristics, OSCAR also supports *yield*-based heuristics, which tend to penalize overall run time much less. This will allow the tester to leverage the strengths of both noise types. It is not yet possible, however, to mix both noise types together.

Another important mechanism, is to allow for probabilistic triggering in noise seeding heuristics. While triggering noise based on a randomized value belonging to an interval can effectively explore a program's state space, probabilistic triggering can offer a more efficient alternative. Instead of having to rely on triggering a very low amount of noise in one thread and a higher on another one, no noise is triggered on the first. Probabilistic noise triggering can also be combined with an interval, for an ideal scenario. OSCAR allows combining both these techniques allowing a tester a lot of freedom in noise seeding configurations.

During the following sections, every type of noise location and heuristic developed will be explained further, detailing the specifics of its implementation. All of the implementations are based on the methods presented and described in Section 4.1 of Chapter 4.

5.3.1 Synchronized Methods and Blocks

The goal set for the first iterations of the OSCAR prototype, was to allow the noising of `synchronized` method invocations and blocks. The noising locations chosen were before and after synchronized method calls and synchronized blocks, which will force additional new interleavings by delaying the execution of the atomic block's routine or the instructions that follow it.

These noising locations were chosen to be tackled first as they are extremely common in Java software, highly prone to high-level data races and its noising instrumentation logic is rather straightforward. The reason for the implementation of the instrumentation logic necessary to support these noising locations not being very difficult, stems from the fact that *Soot*'s Jimple-generated AST's API, comes with a plethora of verifiable flags for statements and variables, which can be used to infer about their properties. More precisely, in the case of methods, a developer can invoke the `isSynchronized()` method, which allows for assessing if a method in the AST is synchronized. In the case of this noising location, it is only necessary to obtain the reference to the method invoked in a given location and check if it is declared as synchronized.

Synchronized blocks require a different approach, being necessary to check if a given instruction is a monitor entrance or exit. These entrances and exits in bytecode directly refer to the bounds of the synchronized blocks in source code.

Since the routines for the implementation of both synchronized method calls and blocks transformer routines did not require more complex analysis techniques, such as callgraph exploration. This made it possible to rely on parallelism provided by body transformers, by inserting it into jtp transformer pack.

5.3.2 Threads

The first two variants of thread-based noise placement locations which were presented in Section 4.1.2, which solely noised the routine of a thread T_a responsible for launching a thread T_b, were fairly straightforward to implement.

The last one, referring to the noising at the beginning of the routine of a launched thread, requires resorting to more complex mechanisms, as it becomes necessary to identify the class, method or lambda expression that is passed as a parameter into the `Thread` object, upon is initialization, and instrument its body. In the case of launching a thread with a previously created `Runnable` class, it is possible to instrument its `run()` method's routine. Luckily, for injected lambdas and functions, as can be seen in [61], and further observed in Listing 5.4, the JVM greatly eases this process, as its compilation process dynamically creates new `Runnable` classes and corresponding methods for these constructs.

The noising logic of this primitive was implemented as a phase for the wjtp pack, as there was a need to resort to a callgraph and explore the bodies that belong to the thread launching (i.e. `run()`) method calls. This essentially means that this transformer's routine is not parallelizable.

One challenge of implementing the thread noising logic, was that it is very usual for thread objects to extend or implement a Java core class such as `Thread` or `Runnable`. This meant it was necessary to create a recursive search which verified every superclass and the implemented

```
void routine() {
    Thread t = new Thread(
        () -> System.out.print()
    );

    t.start();
}
```

(a) Lambda thread call without noise

```
void routine() {
    Thread t = new Thread(bootstrap());
    Controller.noise();
    t.start();
    Controller.noise();
}

// This is dynamically generated
void bootstrap() {
    Controller.noise();
    System.out.print()
}
```

(b) Lambda thread call with noise

Listing 5.4: Noising threads launched with lambdas.

interfaces at each level of the hierarchy. Currently, this check is always being done. A future improvement in performance would be to use some sort of data structure which registered classes which were verified to either extend the `Thread` class or implement the `Runnable` interface. This could make a significant difference in bigger programs with many classes and subclasses.

5.3.3 Reentrant Locks

As can be expected, this noise injection phase's routine is extremely similar to the one which noises synchronized method blocks. During the routine, it is necessary to parse a method body and find instances of method invocations which belong to the `ReentrantLock` Java class. This meant that this noising routine was embarrassingly parallel, allowing it to be inserted in the jtp pack.

An additional improvement which could be made to the current implementation, would be to also verify if the method invocation belonged to a subclass of `ReentrantLock`.

5.3.4 Shared Variables

The current shared variable implementation does not target every single possible type of variable dependency, choosing to target shared fields only, instead. But, since OSCAR will be open-sourced and its architecture was designed with easy scalability in mind, the building blocks created for this algorithm can be further refined and extended.

OSCAR's shared variable algorithm focuses only on verifying dependencies between fields and not every single variable. Additionally, dependencies stemming from conditional statements were not implemented. These include both private and public variants of static and virtual fields. The rationale behind prioritizing fields was that these are, in the case of static fields, shared between all threads, and in the case of virtual fields, can be shared by many threads. As such, they can be expected to be a big source of concurrency errors. Still, extending the current implementation to also support any variable is possible.

The implementation of a noise injection heuristic such as this is a lot more complex than the ones previously presented throughout this section, requiring a more fine-grained approach.

5.3.4.1 Dependency Graph Construction

The process of constructing a dependency graph is, as is to be expected, a transformer which performs a callgraph exploration of the whole program. This recursive exploration iterates through every method body of the program and checks each of its statements for dependencies, adding them to the graph. The parsing necessary to extract every variable referenced in every statement is a very long and harduous process, as it is necessary to account for every type of statement and value. An example of a complex statement, is an assignment whose r-value is a function which takes another function as its parameter. A recursive implementation is thus inevitable, if we wish to extract every single variable which is referenced in a given statememt.

In OSCAR, the collection of all the variable dependencies parsed from the constructed dependency graph takes the shape of a map, where every node is mapped to the collection of its dependencies. This makes querying a lot more efficient when processing assignments in the noising phase.

The current implementation of the shared variable noising heuristic, only concerns itself with dependencies which stem from reads and writes to shared fields, present in assignments. Extending the existing shared variable noising implementation with the ability to detect dependencies stemming from conditionals, should considerably empower OSCAR. However, creating the necessary parsing logic requires a relatively complex recursive search algorithm, which was left for future work.

5.3.4.2 Dependency Graph Flattening

After the DCG is fully populated, this phase recursively explores the resulting directed dependency graph to create a hash map with variables as keys and the list of their dependencies as values. This allows for much quicker access when entering the next phase, which is responsible for checking dependencies and noising.

5.3.4.3 Shared Variable Noising

The last phase of the shared variable noising algorithm is the only one that is parallelizable, and therefore implemented as a method body transformer.

During this phase, every statement in the program is checked for dependencies, extracting the referenced variables, exactly like in the first step, explained in Section 5.3.4.1. After extracting said variables, the dependency map built in the second phase is cross-checked for dependencies. If a dependency exists, noise is inserted before and after the problematic statement.

The shared variable noising could be further optimized by verifying precisely which variable accesses are done concurrently, checking if their respective statements are invoked from threads, and if these threads are ran concurrently. The current technique, while benefiting from its

simplicity, falls short by possibly generating a lot of false positives. This would allow for more efficiency when performing noise injection analysis with OSCAR.

5.4 Program Trace Analysis

Program tracing is one of OSCAR's main features, as a noise injection framework. The entire trace creation and analysis logic is a complex process which involves every single of OSCAR's three components.

OSCAR's trace analysis features were built into a completely separate component. The trace analysis routine is implemented as two Python scripts with several parametrization options.

This section will detail how the trace analysis logic is implemented throughout its three phases. The first being the construction, which is handled by OSCAR's instrumentation engine and run time noise injection library, and the last being handled by OSCAR's trace analysis component.

5.4.1 Trace Construction

The OSCAR instrumentation engine is responsible for inserting noise instructions into heuristically-defined locations throughout a program's code. These noise instructions, as explained earlier, are implemented as calls to the OSCAR Controller — the main component of OSCAR's run time noise injection library.

As such, the tracing is made by piggybacking on the `noise()` calls, which are hardcoded with a UUID as a parameter, to represent the trace location. Inside the `noise()` method, it is then possible to query the thread to get its ID. These two IDs together form a (thread ID, noising location ID) pair with the necessary information to construct a program trace.

However, it only makes sense to trace noising locations which are actually affecting synchronization, meaning inactive noising locations are not traced. A strategy of disabling some noising locations can be used when programs generate exceedingly long traces. This occurs especially when there is one or more tracing location inside a loop.

OSCAR includes an extendable interface for creating implementations for writing the outputted trace files. There are three different implemented mechanisms: console output, file output and lazy file output. Lazy file output differs from regular file output, as it buffers all the data in main memory, before saving it to a file, sacrificing heap space for the added performance of avoiding the slow I/O operations. Writing the trace to a file, resorting to a buffer is ideal, as the program running is not affected by I/O delays. This implementation was made specifically to avoid this issue. A tester must be careful however, as exceedingly large programs may result in an overflow of the available memory. If this happens, two equally bad scenarios may occur: the first is the program crashing due to a Java `OutOfMemoryError`, which signals that the available heap is full. The second is when the memory starts storing data on the hard disk (i.e., swap), severely affecting performance and, consequently, noise injection efficiency and effectiveness.

5.4.2 Trace Interpretation

Program traces are interpreted by OSCAR's trace analysis library. During this step, OSCAR maps the (thread ID, noising location ID) tuples to unique symbols, avoiding the redundant information from the originally very long string representation of a trace location.

Since the symbols need to be unique, an encoding format which allows for a very large amount of characters must be used. The Unicode standard is one such representation format, allowing for over one million unique characters (i.e., symbols). Thus, the mapping $(T_{id}, L_{uuid}) \longrightarrow U_x$, where (T, L) is a trace location tuple, and U_x a uniquely assigned Unicode character, is possible. It is worth noting, however, that the upper limit of around one million unique symbols, could jeopardize the analysis when exploring an extremely large program. After this step, OSCAR's traces are transformed into a series of strings, allowing the use of string comparison algorithms for obtaining an insight into how different these interleavings are from each other.

Still, OSCAR trace interpretation allows for additional modes which will now be described:

- Maintaining the original thread IDs, instead of assigning them a new ID, based on the order of their appearance.
- Completely ignoring thread IDs, when constructing the modified trace. This may be helpful in programs which launch a great number of threads which all do the same routine. In this case, the thread IDs become irrelevant.
- Treating every re-appearance of any given (thread ID, noising location ID) pair as unique, by assigning it an index representative of the order of appearance. This approach ultimately causes every trace point to be mapped to a unique representation.

This script can be configured to run the program a specific amount of times, thus obtaining a series of traces, which can then be interpreted and compared. Another metric that is obtainable during this phase is the program's average run time, from a given amount of runs.

5.4.3 Trace Comparison

The trace analysis library, the same component which interprets the traces produced by OSCAR's noise injection testing, is also responsible for extracting relevant metrics from them, through analysis and comparison. The comparison logic extracts from a series of OSCAR-generated program traces, both of the metrics presented in Section 4.2.3.2. These two metrics are: the number of unique interleavings and the average interleaving distance together with average interleaving distance's standard deviation.

For the average interleaving distance, the following string comparison algorithms can be used: Levenshtein distance [129], Damerau-Levenshtein [48] distance, Jaro similarity [97], Jaro-Wrinkler similarity [197] and Hamming distance [77].

5.5 OSCAR Interface

When designing any framework with the intent of garnering considerable adoption, it is of the utmost importance to provide it with a simplistic interface (e.g., CLI, API or graphical user interface (GUI)) paired with solid documentation.

Every single type of interface, such as configuration files and even GUI's, has its pros and cons, however, a CLI was chosen for interacting with OSCAR, as it was the most versatile. The following sections go over the options both OSCAR Instrumenter and the OSCAR Controller allow.

Both of the OSCAR Instrumenter and OSCAR Controller's argument parsers follow the same logic. The "usage" informs the user how to run the program, together with the positional arguments. A positional argument between less-than and greater-than signs, means that the argument is mandatory, while an argument between square brackets is optional. The options also follow the same pattern: short alias, long alias, option argument format and a short explanation. The following sections will define precisely what each options does. Both sections show, in Listings 5.5 and 5.6, the output of the argument parsers `help` command.

5.5.1 OSCAR's Instrumenter Command Line Interface

Listing 5.5 shows the output of the OSCAR Instrumenter's CLI help command. This command details all of its available run options. Additionally, the reader can find an in-depth explanation, of each of these parameters, below.

```
Usage:
    java oscar <targetfolder | targetjar> <mainclass> <outputdirectory> [oscar_options]
OSCAR options include:
   -vb --verbose    Flag  False     Enable full logging.
   -b  --blacklist  List  [java.,]  Set blacklisted classes by prefix
   -j  --jar        Flag  False     Enable JAR file processing mode.
   -h  --help       Flag  False     Print Help.
   -v  --version    Flag  False     Print Version.
```

Listing 5.5: Ouput of the "help" command from OSCAR's CLI.

targetfolder | targetjar: either the root folder of the Java program, containing all `.class` files, or a target `.jar` file. These files will be fed to OSCAR, so they can be instrumented.

mainclass: the name of the *main* class. Sometimes we may wish to feed another *main* class, other than the one present in the *main* project file, such as for unit testing.

outputdirectory: the directory to which the Java files instrumented by OSCAR, will go to.

oscar_options: this placeholder represents the location where the argument parser expects the options to be located.

verbose: this options makes the logger output additional information, most useful for debugging purposes.

blacklist: accepts a list of class names and prefixes, separated by commas, whose method bodies will not be instrumented by OSCAR. By feeding "java." to this argument, all classes contained in the "java.*" prefix, are blacklisted, while feeding "java.util.logging.Logger", will only blacklist this one specific class.

help: signal the argument parser to output the `help`, found in Listing 5.5.

version: returns the current version of OSCAR.

5.5.2 OSCAR Controller's Command Line Interface

In Listing 5.6, the output of the OSCAR Controller's CLI `help` command is illustrated. A more in-depth explanation of its parameters, can be found below.

```
Usage:
    java <mainclass> [oscar_controller_options] (to execute a class)
  or:
    java -jar <mainclass> [oscar_controller_options] (to execute a jar file)

OSCAR controller options include:
    -a --args                         String   -        Inject arguments into the program
    -co --console-output              Flag     False    Output trace to console
    -fo --file-output                 Flag     False    Output trace to a file
    -lfo --lazy-file-output           Flag     False    Lazily output trace to a file
    -p --noise-probability            Float      1      Probability of noise being triggered
    -M --max-noise-intensity          Long      10      Set maximum noise intensity
    -m --min-noise-intensity          Long       0      Set minimum noise intensity
    -d --disable-noise                Flag     False    Disable all noise
    -dt --disable-tracing             Flag     False    Disable all noise tracing
    -d1 --disable-pre-noise-trace     Flag     False    Disable pre-noise tracing
    -d2 --disable-post-noise-trace    Flag     False    Disable post-noise tracing
    -y --yield                        Flag     False    Set noise type to yield.
    -nc --noise-categories            String[]  {}      Set active noise categories.
    -nl --print-locations             String[]  {}      Set active noise locations.
    -pnl --print-noise-locations      Flag      -       Print available noise locations.
    -v --version                      Flag      -       Print OSCAR version.
    -vb --verbose                     Flag     False    Enable full logging.
    -q --quiet                        Flag     False    Disable logging.
    -h --help                         Flag     False    Print Help.
```

Listing 5.6: Ouput of the "help" command from the OSCAR Controller's CLI.

mainclass: the instrumented program's Java *main* class which will be run.

oscar_controller_options: this placeholder represents the location where the argument parser expects the options to be located.

args: this option allows the injection of arguments into the subject under test (SUT). These arguments should be inside quotes.

console-output: enables the output of the program's noising trace to the console.

file-output: enables the output of the program's noising trace to a file.

lazy-file-output: enables the output of the program's noising trace to a buffer, which will be flushed to a file, when the program finishes.

noise-probability: defines the probability p, with $0 <= p <= 1$, of noise being triggered upon reaching a noising statement.

max-noise-intensity: defines the maximum intensity of noise, in milliseconds, for `sleep` (default), and repetitions, for `yield`.

min-noise-intensity: defines the minimum intensity of noise, in milliseconds, for `sleep` (default), and repetitions, for `yield`.

disable-noise: completely disables noise injection but not the program's noising trace. Essentially sets the noise intensity to 0. This option allows for comparing the impact of the program with and without noise being injected.

disable-noise: completely disables the program's noise insertion, but it may still be possible to trace.

disable-pre-noise-trace: disables tracing before injecting noise into the program.

disable-post-noise-trace: disables tracing after injecting noise into the program.

yield: instructs the controller to use the `yield` noise type, when injecting noise. Currently, `sleep` is the default.

noise-locations: allows a tester to set the list of enabled noise placement locations. An example would be "before a thread routine starts". This argument takes a shorthand code, which can be consulted by running the "print-noise-locations" command.

noise-categories: allows a tester to set the list of enabled noise placement categories. An example would be "thread-based noising". Like "noise-locations", this argument takes a shorthand code, which can be consulted by running the "print-noise-locations" command.

print-noise-locations: this command output a dynamically-generated list of all the implemented noise placement categories and locations. Each one of these is identified by a code, composed by the initials of its heuristic and location name. Its output can be seen in Listing 5.7.

verbose: this option makes the logger output additional information, most useful for debugging purposes.

quiet: the contrary of "verbose", making the logger output less information.

help: signal the argument parser to output the `help`, found in Listing 5.6.

The `print-noise-heuristics` contains information of all of the currently implemented noising locations. This list is dynamically populated, as more locations are added and defined in the OSCAR Instrumenter implementation. The current output of the `print-noise-heuristics` command can be found in Listing 5.7.

```
Available noise placement categoeies:
Category                    Shorthand code
-------------------------   -------------------------
SYNCHRONIZATION BASED        sb
THREAD BASED                 tb
LOCK BASED                   lb
SHARED VARIABLE BASED        svb

Possible noising placement locations:
Category                    Location                          Shorthand code
-------------------------   -------------------------         -------------------------
SYNCHRONIZATION BASED        BEFORE SYNC METHOD CALL           sbbsmc
SYNCHRONIZATION BASED        AFTER SYNC METHOD CALL            sbasmc
SYNCHRONIZATION BASED        BEFORE SYNC BLOCK                 sbbsb
SYNCHRONIZATION BASED        AFTER SYNC BLOCK                  sbasb
THREAD BASED                 BEFORE THREAD LAUNCH              tbbtl
THREAD BASED                 AFTER THREAD LAUNCH               tbatl
THREAD BASED                 BEFORE THREAD ROUTINE             tbbtr
LOCK BASED                   BEFORE REENTRANT LOCK LOCK        lbbrll
LOCK BASED                   AFTER REENTRANT LOCK UNLOCK       lbarlu
SHARED VARIABLE BASED        BEFORE SHARED FIELD ACCESS        svbbsfa
SHARED VARIABLE BASED        BEFORE SHARED LOCAL ACCESS        svbbsla
SHARED VARIABLE BASED        AFTER SHARED FIELD ACCESS         svbasfa
SHARED VARIABLE BASED        AFTER SHARED LOCAL ACCESS         svbasla
MISCELLANEOUS                BEFORE CLASS INITIALIZATION       mbci
MISCELLANEOUS                BEFORE THREAD EXTERNAL FIELD REF  mbtefr
MISCELLANEOUS                AFTER THREAD EXTERNAL FIELD REF   matefr
```

Listing 5.7: Ouput of the "print-noise-locations" command from the OSCAR Controller's CLI.

Chapter 6

Validation and Evaluation

This chapter is divided into three parts. Throughout the first part, the IBM Concurrency Benchmark [80], a notable concurrent software benchmark, is presented, doing an in-depth analysis of each of its programs and the specificities of the concurrency errors contained in them. The programs contained in this benchmark, are the programs which will be used both for the validation and evaluation of OSCAR.

During the second part, the OSCAR implementation will be validated, both functionally and non-functionally.

Lastly, during the third part, OSCAR will be evaluated through a series of tests, with the aim of verifying if OSCAR can be used as a software analysis tool for concurrency error detection, by running these programs with and without noise, in order to compare the amount of triggered erroneous interleavings. Next, a subset of these programs are used for interleaving analysis, to assess if OSCAR is capable of exploring more of a program's state space when injecting noise into a program's routine. Finally, the last test, is a run time analysis comparison, to check the performance penalty that can be expected from testing a program through OSCAR's noise injection analysis.

6.1 The IBM Concurrency Benchmark

For validating and testing OSCAR's noise injection capabilities, the IBM Concurrency Benchmark [80] was used. This benchmark, developed at IBM Haifa, by the developers of ConTest [58], consists of dozens of programs with latent errors and was created with the express purpose of serving as a means of comparing different concurrent software analysis tools, such as the ones relying on noise injection. However, the entirety of the benchmark is not publicly available, as some programs are proprietary, meaning we only had access to 24 of these programs. The majority of these publicly available programs were developed as assignments by students at Haifa University.

6.1.1 IBM Concurrency Benchmark Programs

All of the programs in the benchmark had to be slightly modified, some more than others, especially since most would write their results to a file, making the analysis of their results more cumbersome. Besides routing their I/O to the console, the modifications were very slight and not sufficient to jeopardize the existence of the errors or the probability of encountering them.

The stats about the IBM Concurrency Benchmark programs, shown in Table 6.1, are a mixture of the data contained in [21], with statistics generated resorting to the CCMetrics (Concurrency Code Metrics) tool [175]. It is important to note that some of the errors contained in the programs in Table 6.1, are not really concurrency errors, in the sense that they do not result from improper or lack of synchronization. Some of these errors result instead from programming anti-patterns, and are annotated as such.

Our definition of an anti-pattern is taken from Bradbury and Jalbert [20] as a recurring bad design solution. In the context of concurrent programming, these bad solutions often lead to unintended results.

Table 6.1: IBM Concurrency Benchmark program statistics.

Program Name	Classes	Methods	Statements	Critical Regions	Bug Types
account	3	9	64	4	Atomicity Violation
airlinestickets	1	3	30	0	Low-level Data Race
allocationvector	3	8	81	3	Atomicity Violation
boundedbuffer	5	23	192	5	Deadlock
bubblesort	4	11	82	3	Order Violation
bubblesort2	2	4	43	1	Anti-Pattern
bufwriter	5	12	70	3	Low-level Data Race
critical	2	3	25	0	Deadlock, Low-level Data Race
dcl	4	6	69	2	Stale Value
deadlock	1	4	4	0	Deadlock
deadlockexception	4	14	64	5	Deadlock
garagemanager	3	23	162	7	Deadlock
linkedlist	5	22	96	1	Stale Value
liveness	4	11	60	4	Starvation
lottery	2	8	52	3	Low-level Data Race
manager	3	8	60	4	Deadlock, Low-level Data Race
mergesort	2	15	118	3	Atomicity Violation
mergesortbug	2	9	60	0	Low-level Data Race
pingpong	3	9	27	0	Order Violation
piper	4	7	31	0	Anti-Pattern
producerconsumer	3	14	99	0	Anti-Pattern
shop	3	12	60	4	Anti-Pattern
suns account	2	5	28	0	Stale Value
xtangoanimation	31	154	639	64	Deadlock

In Appendix A, the reader can find a detailed description of how each of these programs

works and the nature of the errors contained in them, based on both the documentation provided by the original developers and further visual code inspection.

6.2 Validation

The objective of this section is to validate OSCAR as a noise injection framework. The nature of these validations are divided into two sections: functional and non-functional. By establishing OSCAR as a valid tool for testing concurrent programs through noise injection, we can proceed to evaluate it.

6.2.1 Functional Validation

The functional testing which will be tackled in the following sections, targets both OSCAR's Instrumenter and Controller, with the intent of making sure that these components are working as intended.

6.2.1.1 Noising Routine Validation

It would make sense that the first functional test would be to check whether the noise statements which OSCAR is instrumenting into the program are both being instrumented into the program and being correctly invoked. A straightforward way to analyse this is to add a counter to each time the controller's noise statement is triggered, and informing the user of how many times the noise statement was called. The results for this test are available in Table 6.2. All programs were ran with the highest pre-defined level of contention.

During these tests, all noise placement locations were active. Table 6.2 shows that all programs had noise statements instrumented into them and that these statements were triggered. Thus, we can consider OSCAR's instrumentation step validated.

6.2.1.2 Noise Placement Validation

The next step would be to validate the noise placement component of noise heuristics. This can be achieved by running one of the above programs with different noise placement settings and checking whether the number of triggered noise locations is the same. The program chosen was the `account` program, with the results available in Table 6.3. Like the previous test, the program was also run under high contention.

The results in Table 6.3, validate that OSCAR's noise location categories are being correctly instrumented into the program. Additionally, the mechanism for choosing combinations of noise placement categories and noise locations are working as intended. This can be concluded from the fact that the amount of triggers per location sums up at the end. The lock-based noise placement category triggered noise count has an expected value of zero, since the program lacks any instance of a lock-based Java concurrency primitive such as `Reentrant Locks`.

Table 6.2: OSCAR instrumentation engine functional validation.

Program Name	Number of noise statement invocations
account	888
airlinestickets	31838
allocationvector	343104
boundedbuffer	4873
bubblesort	940
bubblesort2	6916998
bufwriter	33208
critical	49
dcl	1323346
deadlock	129
deadlockexception	3465
garagemanager	39685260
linkedlist	12
liveness	636
lottery	1356970
manager	29198882
mergesort	860
mergesortbug	3380
pingpong	1454
piper	19986
producerconsumer	4371
shop	2056
suns account	80837
xtangoanimation	281

Table 6.3: OSCAR noise placement functional validation (account program).

Active noise placement categories	Active noise placement locations	Number of noise statement invocations
—	—	0
SYNCHRONIZATION BASED	—	100
SHARED VARIABLE BASED	—	762
LOCK BASED	—	0
THREAD BASED	—	30
SYNCHRONIZATION BASED THREAD BASED	—	130
—	AFTER SYNC METHOD CALL	50
—	BEFORE SHARED FIELD ACCESS	190
—	AFTER SYNC METHOD CALL BEFORE SHARED FIELD ACCESS	240
ALL	ALL	892

6.2.1.3 Noise Seeding Validation

It is now necessary to validate the noise seeding capabilities of OSCAR. This is accomplished by checking if a program responds differently to different noise seeding configurations. As such, we will compare the run time impact of increasing intensity levels of the `sleep` noise type in the `account` program. During this test, only synchronization-based and thread-based noise placement locations were active.

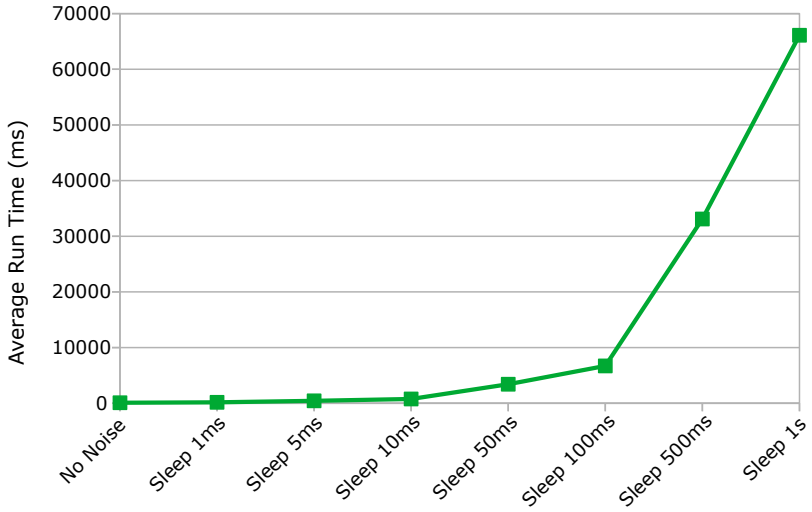


Figure 6.1: Results from seeding validation with the account program.

Figure 6.1 illustrates the effect of the different noise intensity configurations with the `sleep` noise type. As expected, the run time of the program being instrumented is climbing proportionately to the amount of inserted noise. This test validates that OSCAR is inserting both injecting noise into the program and varying its intensity.

6.2.2 Probabilistic Noise Injection Validation

Validating the probabilistic noise injection mechanism can be done in similar fashion to the noise seeding validation, shown earlier in Section 6.2.1.3. This means checking the run time for a program with varying probabilities of noise triggering, and see if the program exhibits a scaling run time, relative to the probability of the noise being injected.

For this test, all of the noise injection placement locations are active and the noise type is set to `sleep`, with an interval of 0-10 ms. Figure 6.2 shows the average run time for each setting of probabilistic noise triggering.

The results show that the increase in run time is proportional to the probability of the noise being triggered. As such, it is possible to deduce that the probabilistic noise triggering mechanism is working as intended.

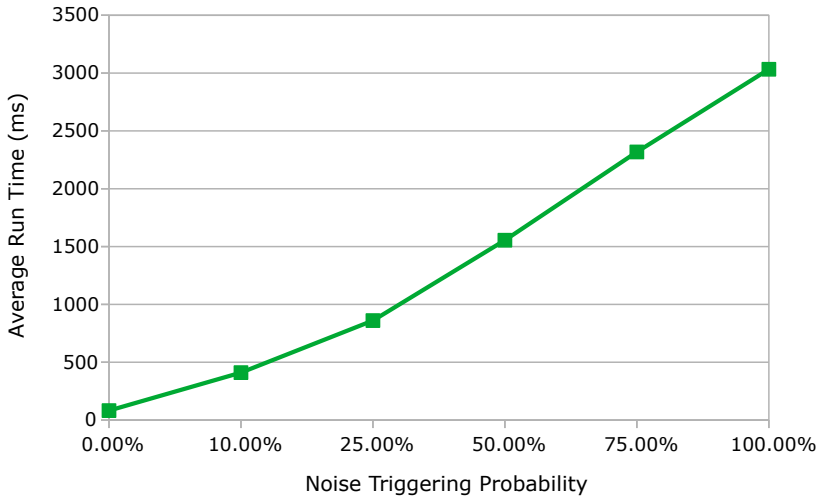


Figure 6.2: Results for probabilistic noise injection with the account program.

6.2.3 Non-Functional Validation

After the functional testing, we will now proceed to address the testing of the non-functional aspects of OSCAR's components. The following sections will test OSCAR's performance. The test bench used for this section, consisted of an AMD Ryzen 7 3700X 16-core processor and 24 GB of RAM.

6.2.3.1 Instrumentation Performance

It is important to assess how long the OSCAR Instrumenter engine takes to instrument a Java program. For this test, a script was created to dynamically generate a given amount of class files, with random names. Each class file had 10 methods with 1000 statements. This meant that each file had a little over 10000 lines of code. Each of the statements were invocations of a synchronized method, meaning it would be instrumented with a noise instruction before and after it.

Figure 6.3 shows how long the OSCAR Instrumentator took to instrument all of the program classes. It is important to keep in mind that, 10000000 statements means a program with 1000 classes, each with 10000 statements, which should be bigger than the vast majority of existing Java programs. Overall, the OSCAR routine is considerably efficient for a program of such size. A tester should expect the instrumentation process to take less than a minute in modern hardware.

6.2.3.2 Noise Injection Overhead

For a program to be tested, the OSCAR Controller has to be bootstrapped to its main method. Additionally, the program has to pass control of the ongoing routine to the Controller, when certain statements (e.g., noising statements) are reached. As such, it is important to ascertain how

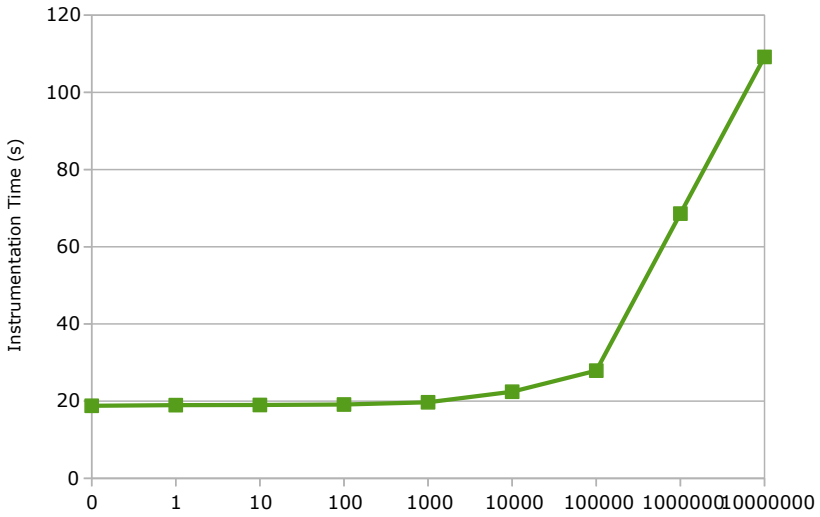


Figure 6.3: OSCAR instrumentation time.

much the OSCAR Controller's logic is affecting the program's overall run time. As such, this test will compare the average run time of a program when uninstrumented and when instrumented by the Controller, but without any noise being injected into it. A large difference is a very bad sign, because it most probably will mean that the noise injection routine will be affected by the added delays, possibly also affecting the effectiveness of the state space exploration.

A small program was created for this specific test, shown in Listing 6.1. The program takes a number of iterations as a parameter, representing the number of times a noise statement will be invoked. Since `doNothing()` is a synchronized method, after being instrumented by OSCAR's Instrumenter, the program will contain a noise statement in the commented location, which will be called the number of times `iterations` is set to. Since noise is disabled, it is possible to understand how much of an overhead the whole logic, necessary to support noising at run time, adds to a program's routine.

```
public static void main(String[] args) {
    int iterations = Integer.parseInt(args[0]);

    for (int i = 0; i < 100000000; i++) { }

    for (int i = 0; i < iterations; i++) {
        // Noise statement will be added here
        doNothing();
    }
}

static synchronized void doNothing() {
    return;
}
```

Listing 6.1: Noise injection overhead test program.

The results for the noise injection overhead are can be visualized in Figure 6.4. As expected,

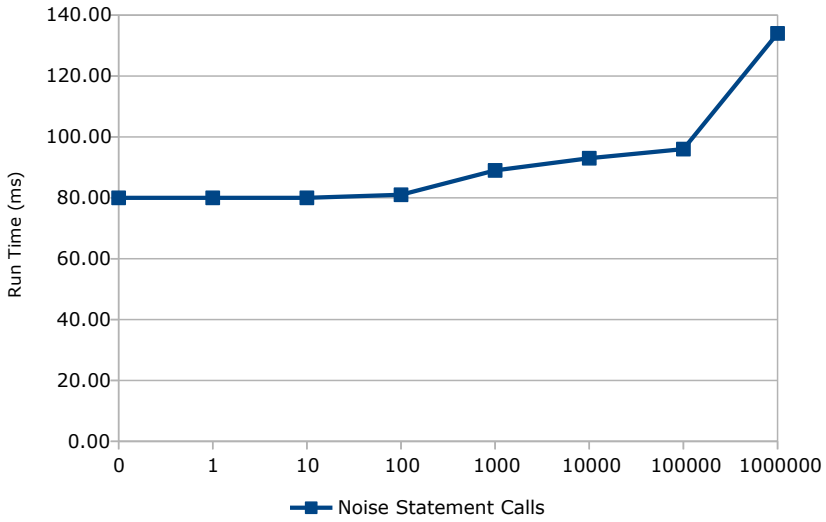


Figure 6.4: OSCAR instrumentation time.

there is an added overhead of running the noise injection logic. This overhead gets more significant after 100000 invocations. From the graph, we can deduce that each invocation will add an overhead of around $\frac{56\,\text{ms}}{1000000} \approx 0.05\,\text{s}$. However, in a multicore environment, the effect of this value can be mitigated, by splitting the overhead over the various threads.

6.3 Evaluation

In this section, OSCAR's functionalities will be evaluated through the analysis of the interleaving information obtained from batch testing a series of programs. The test bench consisted of a Linux KVM instance, limited to a single core, for the majority of tests and unless otherwise mentioned, and 4 GB of RAM, running Ubuntu 22.04, on an AMD Ryzen 7 3700X 16-core processor and 24 GB of RAM. The choice of testing OSCAR using a single-core system, was due to the necessity of creating an environment where the *yield* primitive, which, by its inherent nature, can behave non-deterministically, would surely have some effect. This choice of testing a noise injection framework with a single core was also made in [58], following the rationale that any concurrent error manifested via noise injection on a single core would also manifest itself under a multicore environment.

To facilitate the batch testing of the instrumented programs and to interpret their respective outputted coverage information, Python scripts were created. These scripts allow for choosing the number of times one wishes the program to be run and which arguments to feed it, automating every process from compilation to output processing. Lastly, in its coverage analysis phase, it allows the tester to verify the implemented coverage metrics.

For every test, unless mentioned otherwise, OSCAR instrumented programs were run with noise before the start of a thread's routine, before and after accesses to shared fields, before and after reentrant locked regions and before and after synchronized regions and method invocations. The very few exceptions were, for example, cases where noising occurred inside a busy-wait

loop, meaning the program's overall run time would become very long. The minimum noise intensity was $1\,\mathrm{ms}$ or 1 iteration, with the maximum being $5\,\mathrm{ms}$, $10\,\mathrm{ms}$, and $25\,\mathrm{ms}$ for the sleep primitive and 5, 10, 25 and 50 iterations, for the yield primitive. An additional final run was done without any noise, allowing for the comparison of OSCAR's efficiency at error triggering with and without noise injected into the SUT's routine. The noising locations before and after launching threads were disabled, as they have a tendency to serialize execution. This effect makes state space exploration much harder in programs where threads with similar routines are sequentially started (e.g., in a loop), while simultaneously sharply rising the program run time.

During these tests, all of the program traces were generated by tracing noising locations immediately before triggering noise. This was due to wanting the program trace to reflect (as close as possible) the order of arrival to a given relevant synchronization location and not the order of finishing the location's triggered noising routine. Another possibility would be to trace before and after reaching a synchronization location (i.e., noise instruction). This would, however, result in longer traces with many redundant entries, affecting the interleaving similarity measurements.

Each program was ran 250 times for each configured heuristic. The vast majority of the tested programs allowed for varying levels of concurrency ("low", "medium" and "high"), with some containing default values, chosen by the original developers. In some cases, no default settings for the amount of contention were provided and, as such, had to be chosen by the author. Lastly, other programs only contained one single concurrency setting, henceforth referred to as the "default" concurrency setting.

6.3.1 Error Detection

During this Section, the results from instrumenting all the presented IBM Concurrent Benchmark with OSCAR and, afterwards running them with and without noise are presented. From these results, it will be possible to ascertain OSCAR's efficacy at triggering the latent errors contained in these programs.

The IBM Concurrent Benchmark programs were designed with error testing in mind, outputting a message symbolizing either success or failure, after finishing their routines. As such, the process was automated by creating a parsing mechanism able to count instances of either successes or failures, and check the program's output after testing it with noise injection.

6.3.1.1 Discussion

The error detection tests involved a total of 24 programs from the IBM Concurrency Benchmark. These program's varied significantly in size, routine, levels of contention and types of concurrency errors. Some of the tested programs were not ideal for testing, given they required features not yet supported by OSCAR. Additionally, given that some programs contained errors which did not stem from improper implementation of concurrency primitives and patterns but simply general programming anti-patterns, a noise injection tool, such as OSCAR, could not improve the rate of their triggering. Errors of this kind are best left to specialized Java static analysis tools, such as Java PathFinder[191] or Java RaceFinder [107].

Given that every single one of the IBM Concurrency Benchmark programs was analysed, the produced graphs and further analysis spanned over 12 pages. As such, the results were moved to Appendix B, available to a reader who wishes to consult a more in-depth analysis of the results.

Of all the results, it is important to highlight the error detection results obtained from analysing the `manager` program, available in Section B.16 of Appendix B. In this program, OSCAR's noise injection routine managed to find an additional latent bug which was undocumented by the original developer.

Overall, the results are very interesting and positive, serving to validate OSCAR as a noise-injection-based concurrent software tool. In 18 of the 24 tested programs, OSCAR managed to expose a latent bug, or raise the probability of the error occurring. Given that this consists of three quarters of the programs, it becomes evident that OSCAR can be used to force a program to trigger erroneous interleavings. OSCAR is, therefore, for the majority of the tested cases, a valid concurrency error detection tool.

In four of the tested programs results, OSCAR managed to trigger different interleavings, by triggering non-erroneous interleavings in program's which exhibited an error in virtually every run, when run without noise. These results show that OSCAR managed to explore more of the program's state space.

One of the programs, `deadlockexception`, had a concurrency error that did not react to noise injection, as it relied on a probabilistic function to occur.

Lastly, two of the tested programs exhibited concurrency error patterns incompatible with OSCAR's currently implemented noise injection heuristics. Still, in one of these programs, OSCAR still managed to trigger an erroneous interleaving, exposing the latent bug.

A very interesting observable phenomenon was that, in some programs, the `sleep` noise type did not do as well as the yield, when it comes to triggering erroneous interleavings. This can be attributed to the fact that the `sleep` noise type, being much more intensive than the `yield`, has the potential to distance locations which could result in errors if executed concurrently. Still, even if these errors are masked, the `sleep` noise type should still be expected to explore more of the program's state space, which is vital, as a program can contain other latent errors. This observation serves to further highlight the importance of testing different noise injection heuristics and configurations in programs, formalized as the TNCS [93] problem.

6.3.2 Interleaving Generation

From all of the programs previously tested for error detection, a smaller number of them had to be chosen to test OSCAR's interleaving generation capabilities. The reason for not testing every single program was, mainly, that many of the programs had implemented similar concurrency programming patterns, resorting to identical mechanisms (e.g., Producer-Consumer through Java `Threads`), which in turn meant that they exhibited similar results in error detection and would, most certainly, exhibit similar results in interleaving comparison tests. Additionally, in its current state, the interleaving comparison algorithm is quadratic and thus still very inefficient and timely.

The prerequisites that the IBM Concurrency Benchmark programs chosen for this test had to fulfil were the following: they should vary in the types of the concurrency primitives used, or at least in how these coordinate; they should not deadlock, as this would not allow us to obtain a full trace and have a fair comparison; they should be programs which showed some reaction to OSCAR's noise injection, negative or positive, meaning it is possible to be sure that OSCAR's noise injection statements are being called and OSCAR is effectively doing something; some programs should show different levels of effectiveness of the `sleep` and `yield` performance. With these requirements in mind, three programs were picked: `account`, `pingpong` and `lottery`.

These tests were run with the objective of verifying OSCAR's effectiveness at exploring the state space of concurrent programs, by comparing the rate of the triggering of additional interleavings, with and without noise being introduced into the tested program's routine.

Still, error detection will always depend more on the program structure and the use of correct heuristics, meaning it is not the best indicator for the efficacy of a noise injection tool. A noise injection tool (or heuristic) should be measured instead by its capacity to explore more of a program's state space.

6.3.2.1 The account Program

The `account` program error detection results, shown in Section B.1, exhibited that a latent bug, undetectable without noise, was exposed by the `sleep` noise type much more frequently than with the `yield` noise type. When observing the amount of unique interleavings generated by each of the noise seeding settings, in Figure 6.5, the reason why, becomes apparent. Every single interleaving generated by OSCAR, using the `sleep` noise type, was unique, meaning there was a much larger state space exploration, in comparison to other noise types.

Coincidentally, the average interleaving similarity, shown in Figure 6.6, is also as expected. Although there is a high degree of similarity between all noise types, the `sleep` noise type interleaving similarity values are consistently significantly lower, with the standard deviation not varying significantly between noise types. After examining trace files of the `account` program, the program trace pattern is as follows: a long single-threaded sequential routine, followed by a concurrent phase and ending with another long single-threaded sequential routine. Since the main thread takes up so much of the trace, big differences in the interleaving seem smaller.

Overall, OSCAR managed to trigger many more, and significantly more asimilar, interleavings, in the `account` program.

6.3.2.2 The lottery Program

The `lottery` program's concurrency error was almost always triggered, regardless of noise, as shown in Section B.15. However, when using the `yield` noise type at a low concurrency setting, the error was masked.

By examining the number of unique interleavings generated by the program, in Figure 6.7, the deficit in error detection exhibited by the `yield` noise type at a low concurrency setting, coincides with the lower amount of different interleavings triggered. It is also important to

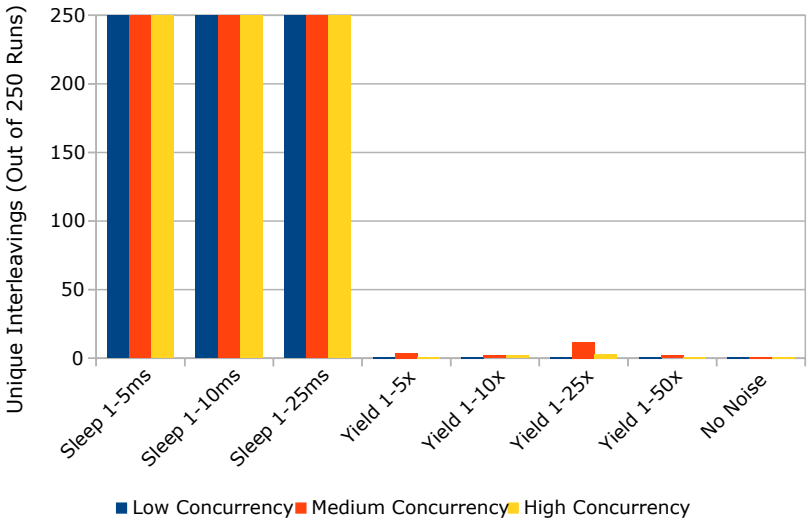


Figure 6.5: account program unique interleavings.

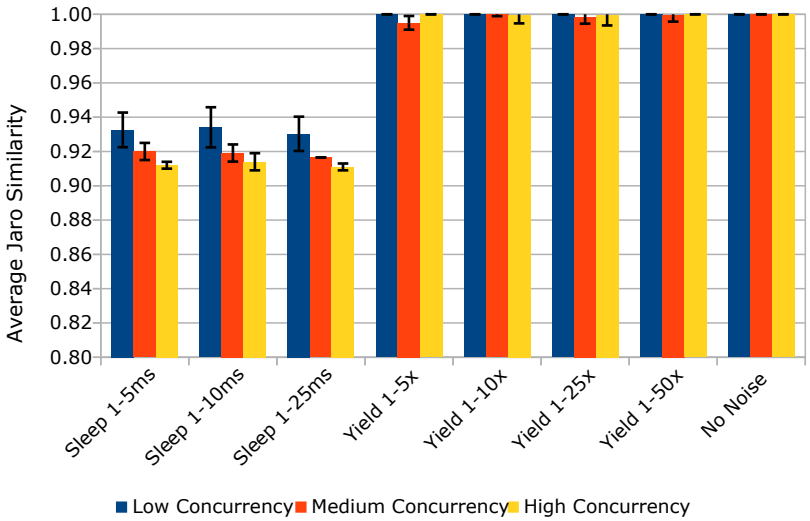


Figure 6.6: account program interleaving similarity.

highlight how the standard deviation in average interleaving similarity is much lower for the `sleep` noise type, meaning `yield`'s state space exploration is not as balanced. Interestingly, while the unnoised program triggered even less unique interleavings, it managed to trigger the most erroneous interleavings, further proving how noise injection can mask errors.

When looking at the average similarity between interleavings, in Figure 6.8, the `sleep` noise type was shown to produce consistently less similar interleavings.

6.3.2.3 The pingpong Program

The `pingpong` program error detection results, shown in Section B.19, showed the exact opposite of the results from the `account` program, shown in Section B.1. In the `pingpong` program results, the bug was not latent at all, triggering with every run, except when certain settings of `sleep`-based noise seeding were used.

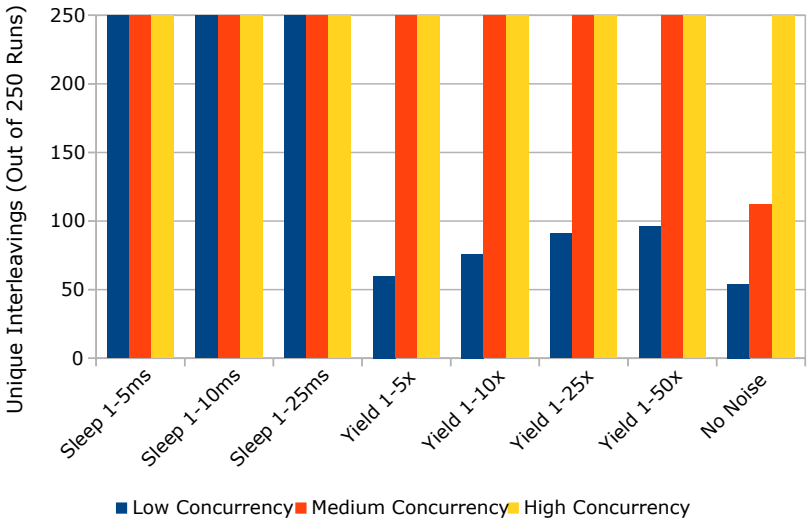


Figure 6.7: lottery program unique interleavings.

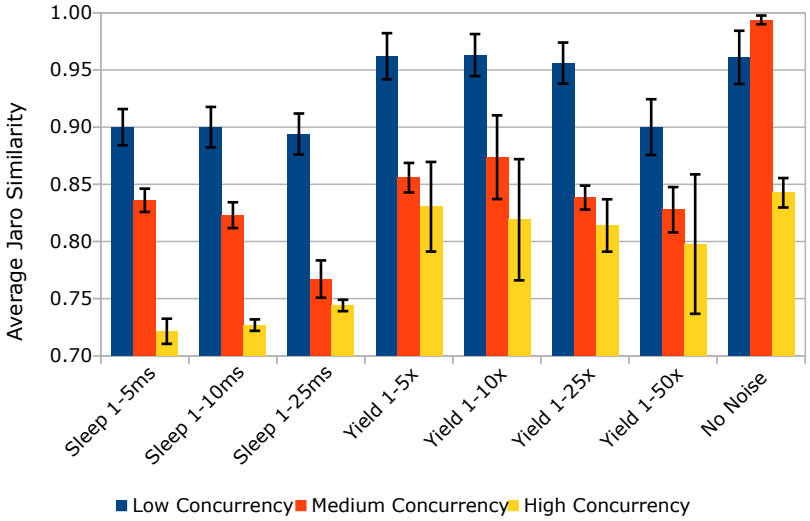


Figure 6.8: lottery program interleaving similarity.

However, when examining the amount unique interleavings, in Figure 6.9, it is possible to witness that all the interleavings triggered by every noise seeding setting are unique. Thus, the program, when noised by `OSCAR`, is exploring a much larger chunk of the state space.

But, the `sleep` noise primitive lower error triggering results are justified when examining the interleaving similarity results in Figure 6.10. The `sleep`-based noise seeding settings managed to produce interleavings which were significantly less similar. This lower level of similarity justifies the `sleep` noise type's ability to trigger non-erroneous interleavings.

6.3.2.4 Discussion

The results have shown that, as expected, the `sleep` noise type has a consistently better capability of exploring more of the program's state space. While on one hand, a tester must be careful, as

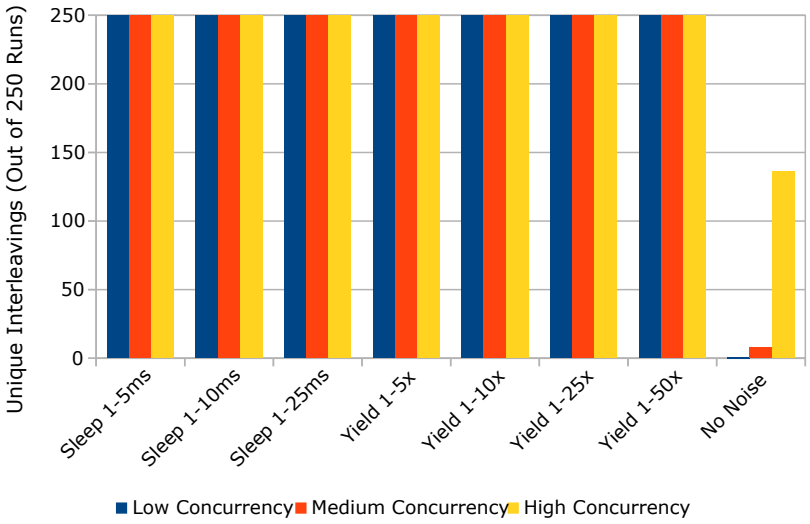


Figure 6.9: pingpong program unique interleavings.

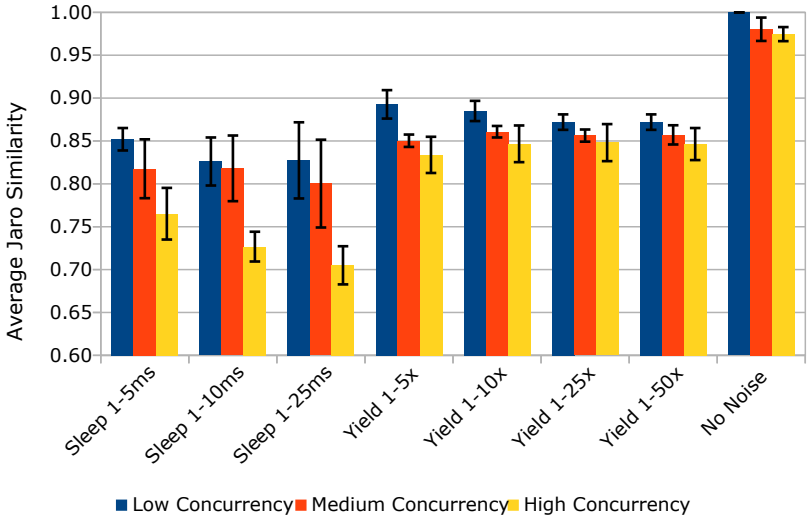


Figure 6.10: pingpong program interleaving similarity.

this feature can mask some errors which are not latent, on the other hand, a tester can be surprised by the `sleep` noise type's higher capability of finding unexpected latent errors.

While the total amount of program runs was set to 250 runs, which bounds the maximum amount of unique interleavings triggered to this very same value, it is quite possible that, given a much higher amount of runs, it would be possible to observe differences between all the noise seeding techniques that managed to trigger 250 unique interleavings in the tests. A technique in which tests are ran multiple times with more and more runs, until diminishing returns are observed, would allow for a tester to obtain an idea of when either a noise seeding technique is reaching its limit, or the totality of the explorable state space is close to being fully covered.

6.3.3 Run Time Analysis

While it is important to understand the efficacy of noise injection in general, and of noise types more specifically, their efficiency is also an important metric. While a `sleep` noise seeding setting bounded by a gargantuan interval would be able to capture every single possible interleaving, given enough time, the amount of time required would be infeasible. As such, for real world usage, it is necessary to leverage a noise injection heuristic's efficiency and efficacy.

The run time analysis tests were made with this objective in mind. After obtaining a metric of effectiveness for various heuristics from the previous interleaving generation tests, these run time analysis tests will allow us to obtain a metric of efficiency for these very same heuristics.

The programs which were chosen for running run time analyses tests had to be the same programs as in the interleaving generation results, under the same exact configurations, since, as mentioned earlier, it was necessary to compare heuristics efficiency and effectiveness.

6.3.3.1 The account Program

The run time analysis for the `account` program, shown in Figure 6.11, shows that, while the `yield` noise type had virtually no impact on the run time, the `sleep` noise type had a significant penalty to the overall run time. Additionally, the penalty imposed by resorting to using the `sleep` noise type grew linearly, as the `sleep` noise's intensity (interval) was increased.

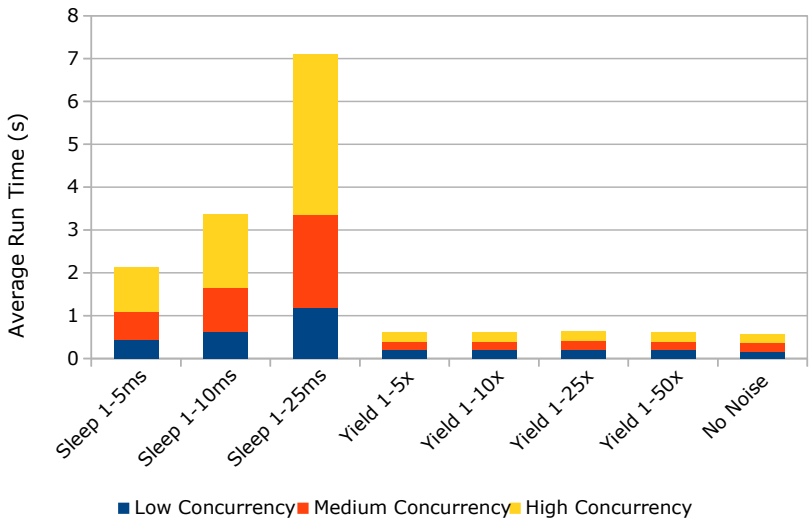


Figure 6.11: account program average run time.

Additionally judging from the interleaving generation results for the `account` program, shown in Section 6.3.2.1, there were no discernable gains from the higher intensities of the `sleep` noise type, while incurring a very steep performance penalty.

6.3.3.2 The lottery Program

In the lottery program, the run time analysis results, depicted in Figure 6.12, show that there is a much lower penalty for using the `sleep` noise type (as opposed to `yield`), in direct comparison

with the run time analysis results of the `account` program, in Section 6.3.3.1. Thus, the use of the `sleep` noise type is especially advisable in this specific scenario.

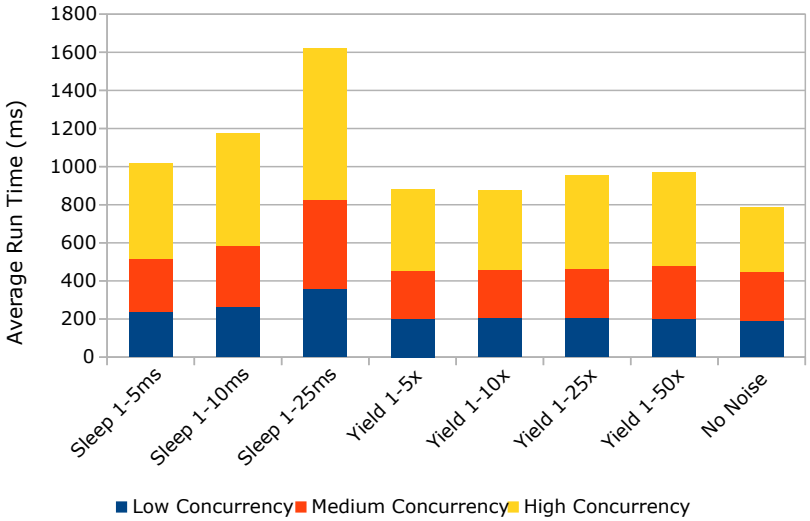


Figure 6.12: lottery program average run time.

6.3.3.3 The pingpong Program

The `pingpong` program results, present in Figure 6.13, show that there is a considerable penalty to using the `sleep` noise type. As such, taking into account the generated interleaving analysis results, presented in Section 6.3.2.3, the use of the `sleep` noise type, with its inherent performance penalty, cannot be justified.

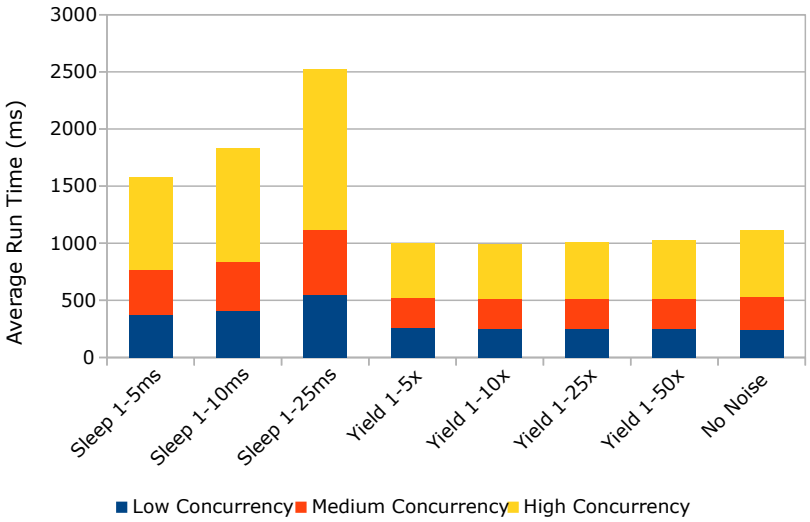


Figure 6.13: pingpong program average run time.

6.3.3.4 Discussion

While it can be expected of the `sleep` primitive to have a more significant performance penalty that `yield`, this penalty does not always equate proportionally to an increase in state space exploration effectiveness. The `yield` primitive, in many instances, can be considered as a valid, highly efficient alternative, more suitable to real-world scenarios.

Since the performance of the noise type depends on so many variables, such as program structure, simultaneous thread count, intensity, and the underlying heuristic, it is impossible to create a general rule as to what should be used, a conclusion which was also reached in [63]. Instead, testers should do a series of smaller tests, alternating parameter, to try and find out which combination of configurations will yield the best results. A mechanism that automates this process could prove to be extremely useful, even more so when empowered with machine learning capabilities. Additionally, a program structure analysis tool like CCMetrics [175], could extract a lot of valuable information from the program, such as the types of concurrency patterns which are most likely to appear. Leveraging a tool such as CCMetrics, as a form of pre-analysis, could eliminate much of the guesswork out of trying to discover the best noise injection heuristic (or heuristics) for exploring the program state space, triggering more and less similar interleavings. This approach would mean the tester (or his tool) would only need to worry about tuning the noise seeding configuration.

6.3.4 Probabilistic Noise Injection Comparison

All the tests until now deterministically injected noise into a program, with an intensity bounded by an interval. Although demonstrably very effective, as could be seen in previous tests, this approach is very inefficient. This test will be responsible for testing the probabilistic noise seeding mechanism. Since the `sleep` noise type is usually inefficient, a probabilistic noise seeding mechanism could help mitigate this effect. This test will allow us to ascertain if it is possible to maintain or even improve the effectiveness of state space exploration, while significantly improving the overall efficiency of testing. While the probabilistic noise seeding mechanism would also make `yield`-based heuristics more efficient, the gains would never be as accentuated as those in `sleep`-based heuristics.

The programs chosen for comparison, were also the `account`, `lottery` and `pingpong` programss. These previous tests have shown that, from all of the noise seeding configurations, the `sleep` noise type behaved the best. Additionally, not much was gained from higher intensities. As such, only the `sleep` noise type will be tested, with its noise intensity set to the interval of 1-10 ms and the noise triggering probabilities to be tested being 10%, 25%, 50%, 75% and 100%.

6.3.4.1 The account Program

The first program to be tested was, like before, the `account` program, with the results available in Figures 6.14 and 6.15. The `account` program exhibited more unique interleavings and a lower

average similarity, as the noise triggering probability increased. However, at 25% noise triggering probability, in average and high concurrency, all variants were able to trigger 250 unique interleavings. This means that, for a similar efficacy, the 25% variant was much more efficient.

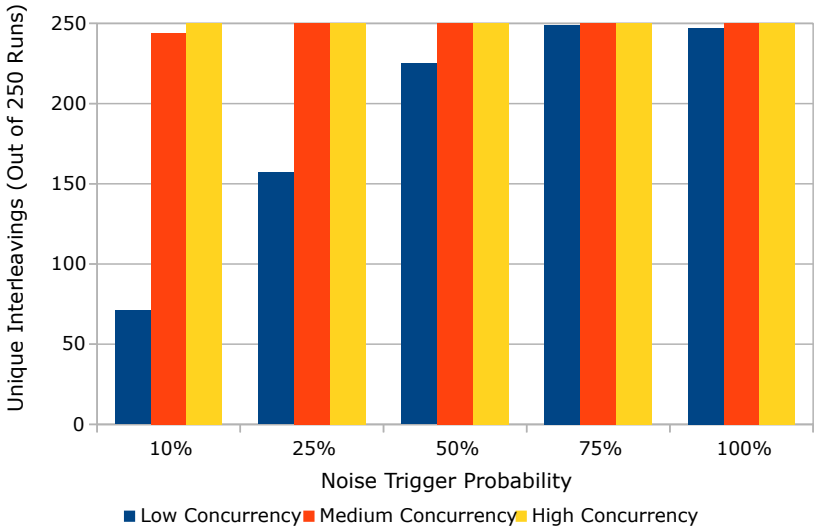


Figure 6.14: account program interleaving generation with probabilistic noise seeding.

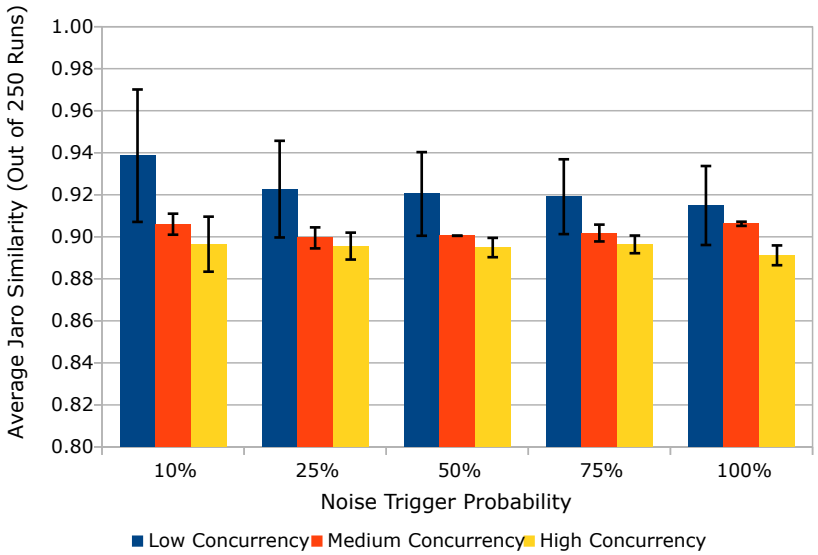


Figure 6.15: account program interleaving similarity with probabilistic noise seeding.

The `account` program was already demonstrated, earlier in the Section 6.3.2.1, to be highly sensitive to the `sleep` noise type. As such, it was somewhat expected of the program to not do as well, when leveraging probabilistic noise triggering. Still, the differences in effectiveness were not very significant, undeserving of the much higher testing time.

6.3.4.2 The lottery Program

The next program to be tested, was the `lottery` program. The results for this program are available in Figures 6.16 and 6.17.

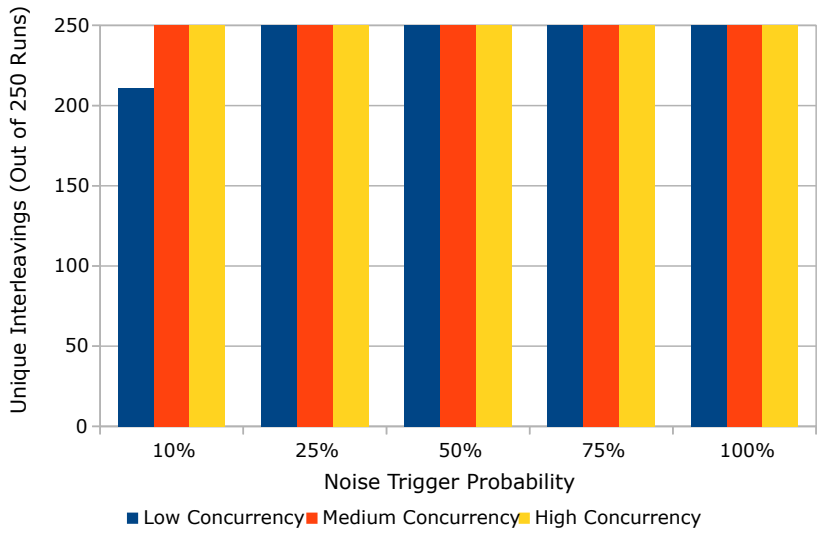


Figure 6.16: lottery program interleaving generation with probabilistic noise seeding.

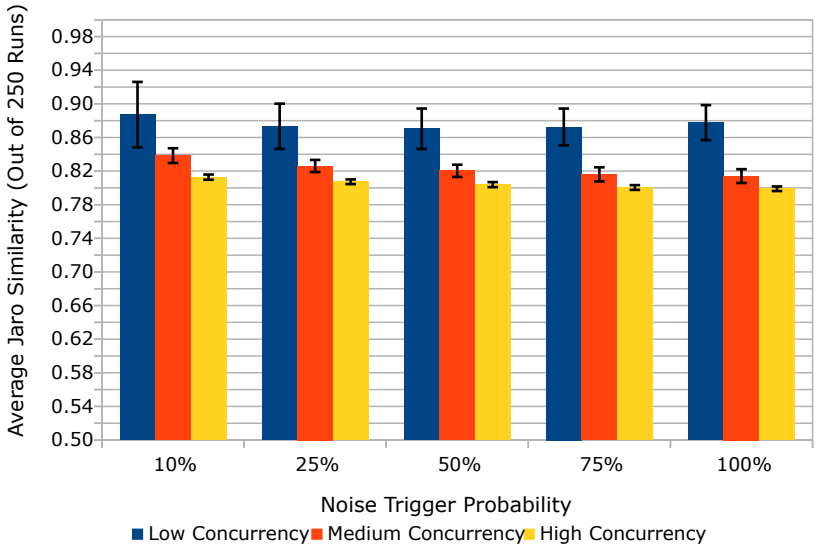


Figure 6.17: lottery program interleaving similarity with probabilistic noise seeding.

The `lottery` program managed to trigger an equal number of interleavings, starting at 25% probability of triggering noise. Additionally, the probabilistic noise generation managed to show basically equivalent performance in generated interleaving similarity, with very small increments in performance with each jump in probability.

This is a great program to test with probabilistic noise injection, as it is possible to obtain a comparable efficacy to deterministic noise triggering, in a quarter of the time.

6.3.4.3 The pingpong Program

The very last of the three programs to be tested with probabilistic noise triggering, is the `pingpong` program. Its results are available in Figures 6.18 and 6.19.

While in previous programs probabilistic noise triggering managed to offer close or similar performance, in the `pingpong` program, it managed to offer even better performance in the average distance of generated interleavings.

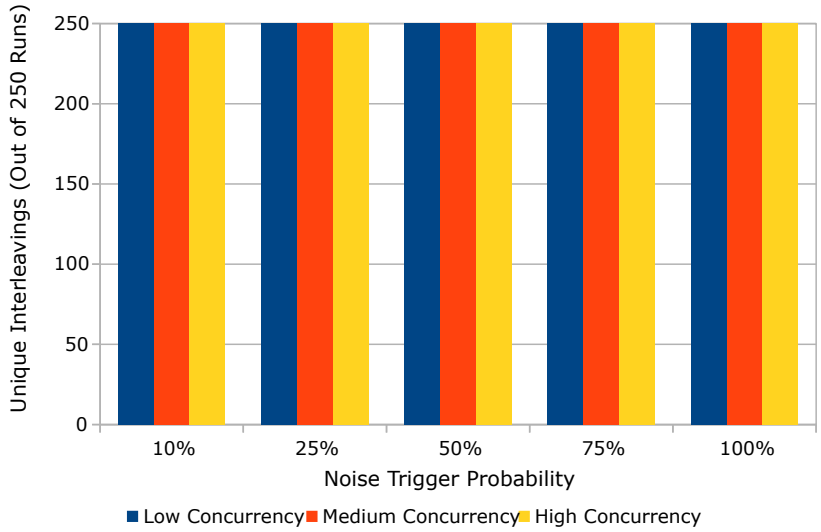


Figure 6.18: pingpong program interleaving generation with probabilistic noise seeding.

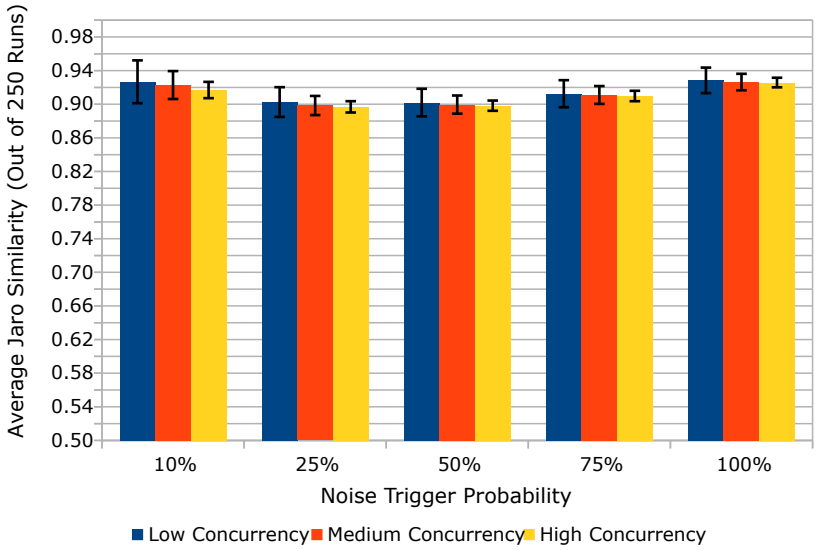


Figure 6.19: pingpong program interleaving similarity with probabilistic noise seeding.

6.3.4.4 Discussion

Probabilistic noise triggering would be an incredibly desirable feature, if it could perform equally or, at the very least, similarly. If so, this would mean that a tester could essentially slash testing times in half or more. A feature which would be extremely useful when testing under the `sleep` heuristic, as its high run time overhead could be significantly mitigated.

From the probabilistic noise injection results, we were surprised that, not only did this feature do almost as well in some cases, it actually did better in others. This validates probabilistic noise injection as a viable technique for noise-injection-based testing of concurrent programs. Additionally, it is important to mention that, although in some cases, probabilistic noise injection may do marginally worse, it can be ran many more times in the same time frame, mitigating or completely eliminating this major con.

Finally, the results demonstrate that the 25 % noise triggering probability offered the most attractive balance between time and performance.

6.3.5 Single vs. Multi-Core Yield Performance

All the previous tests were done resorting to a single-core test environment, with the intent of ensuring the `yield` noise type had an effect. While this conjecture is sound, it remains necessary to validate if increasing the amount of cores on the machine, actually affects the actual performance of the `yield` noise type. As such, this section will compare the efficacy of the state space exploration of the `yield` noise type, with different amounts of available cores.

It only makes sense, however, to examine the interleaving generation penalty on programs where the `yield` primitive managed to generate a significant amount of interleavings, when in a single-threaded environment. As such, the program tested during this section must have managed to respond well to the `yield` primitive, during the previous tests. Additionally, the performance with different core counts will only be compared when running at a "high" concurrency setting, as in a low contention setting, the additional threads may not even be necessary for the program. Lastly, it is important that the tested program has a routine in which multiple threads are active simultaneously and not just 2 or 3, or else the results will also not be useful, as the `yield` primitive will not be doing anything but adding a very tiny delay at each noising location.

The programs which were chosen for this test were `lottery`, `bubblesort` and `mergesort`.

6.3.5.1 The lottery Program

Given the sheer amount of threads (hundreds), which are launched during the `lottery` program's routine, the resulting state space is incredibly large. As such, it is no wonder that all different core counts managed to trigger 250 unique interleavings out of the 250 runs, as shown in Figure 6.20. Still, it is important to note that this shows that the added core count did not affect the `yield` noise type in this specific scenario.

More interesting, is the average interleaving similarity, which seems to actually improve as the core count grows, as shown in Figure 6.21.

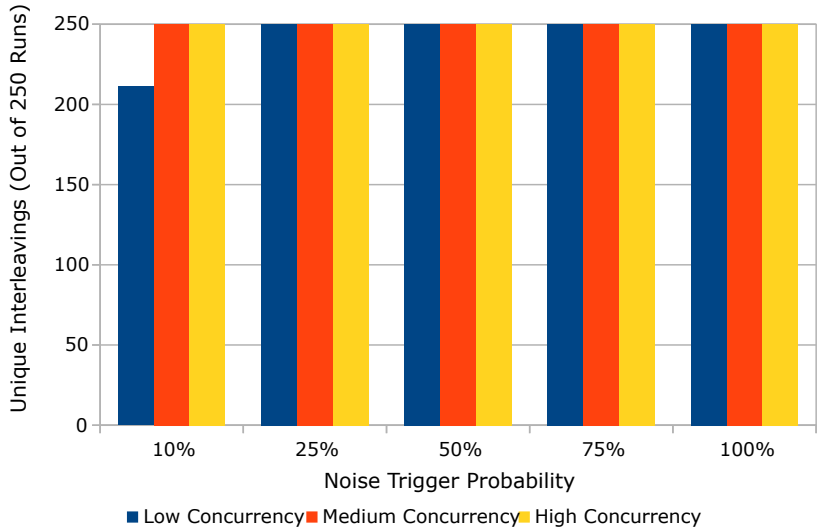


Figure 6.20: lottery program number of unique interleavings with yield and varying core counts.

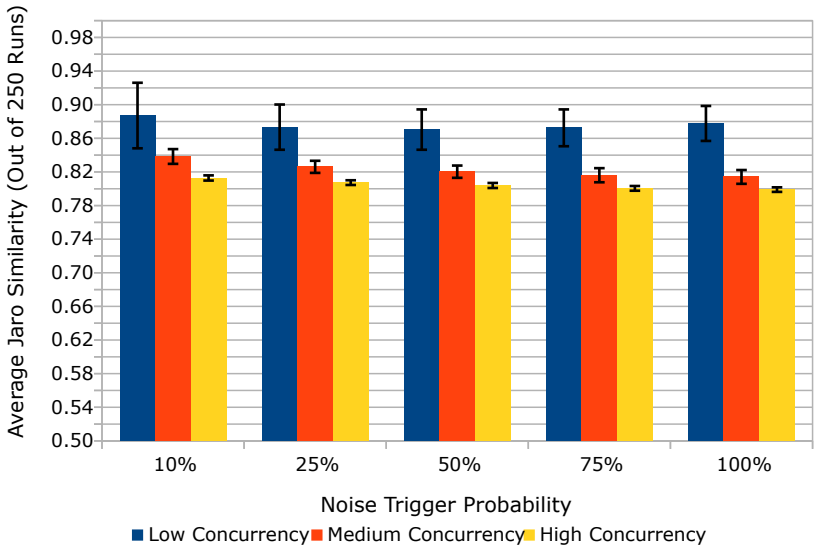


Figure 6.21: lottery program average interleaving similarity with yield and varying core counts.

6.3.5.2 The bubblesort Program

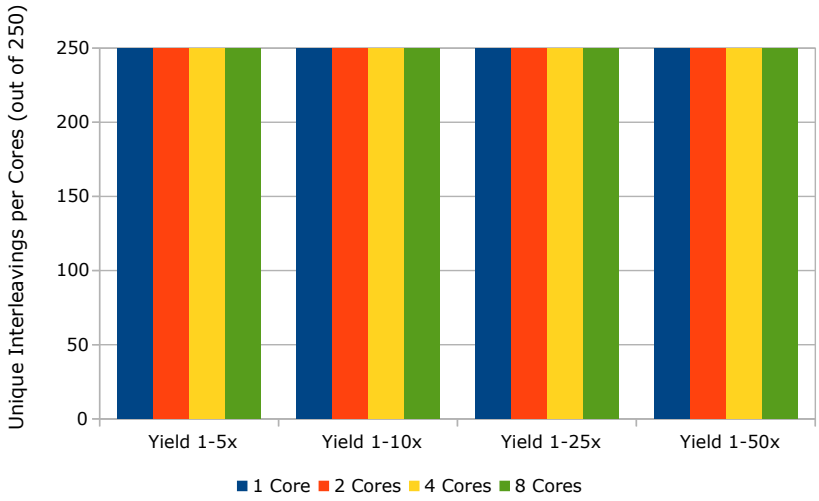


Figure 6.22: bubblesort program number of unique interleavings with yield and varying core counts.

Like the previously tested `lottery` program, all 250 interleavings were unique in 250 runs with every single core count, in the `bubblesort` program tests. These results are illustrated in Figure 6.22.

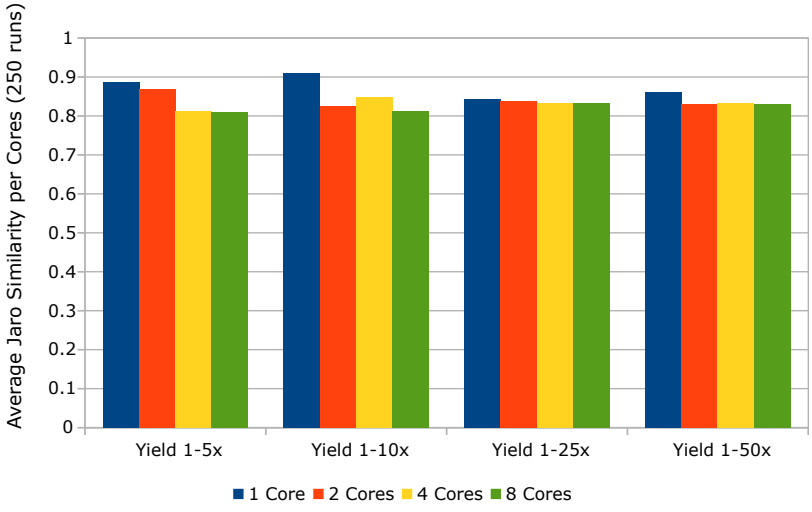


Figure 6.23: bubblesort program average interleaving similarity with yield and varying core counts.

Figure 6.23 illustrates the interleaving generation results for the `bubblesort` program. Unlike the previous `lottery` program, the gains from an increasing core count were not very significant after 2 cores.

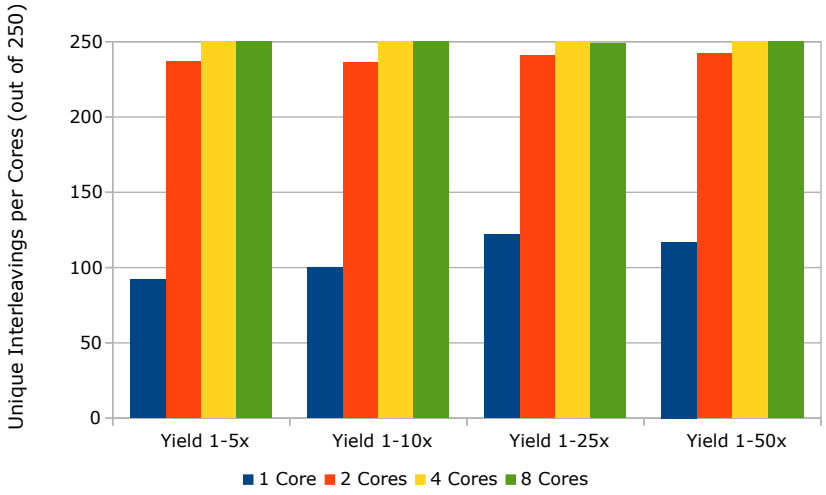


Figure 6.24: mergesort program number of unique interleavings with yield and varying core counts.

6.3.5.3 The mergesort Program

For the `mergesort` program, the unique interleavings generation results are very different from previously tested programs and very interesting. These results, illustrated in Figure 6.24, demonstrate that, at higher core couts, the `mergesort` program had significantly more unique interleavings triggered. This can be attributed to a smaller state space, compared to previous programs, and serves to prove that `yield` does not always work better in a single core environment.

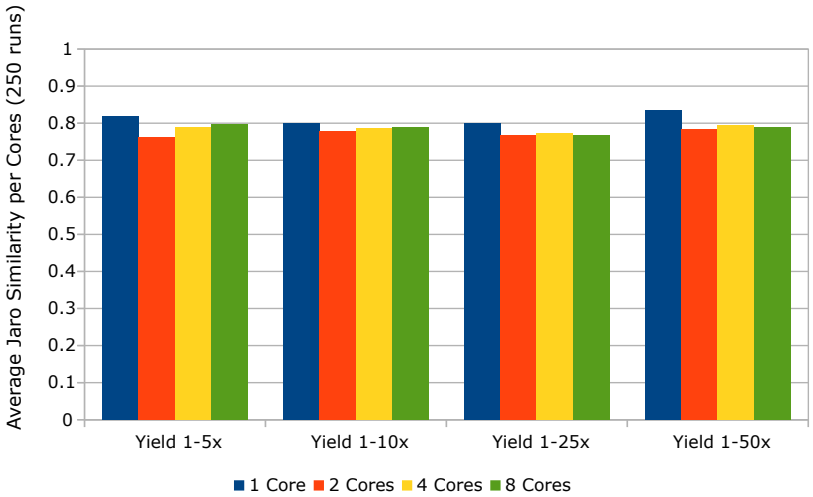


Figure 6.25: mergesort program average interleaving similarity with yield and varying core counts.

In Figure 6.25, the interleaving generation results for the `mergesort` program are shown. Although there is a clear improvement, when jumping from 1 to 2 cores, the results from 2 to 8 cores do not show significant variations.

6.3.5.4 Discussion

Together, the results obtained from the three tested programs demonstrate that the `yield` primitive can be effectively used in a multicore environment. Additionally, the `yield` noise type may even show better performance when in a multicore environment. This is an important insight since if this were not so, testing with the `yield` noise type would significantly increase testing run times, meaning the `yield`'s reduced run time impact could not be enough to make it more efficient than the `sleep` noise type.

Our results show that the `yield` noise type can be confidently used in a, much more performant, multicore setup, resulting in a much higher testing efficiency by exploiting program parallelization.

Chapter 7

Final Discussion

7.1 Conclusion

During the course of this dissertation, we have made an in-depth study of the state of the art in noise injection, contextualized the noise injection problem in the Java programming language and have proposed OSCAR, a new noise injection framework for the Java programming language. Furthermore, we proposed both a new taxonomy for categorizing noise injection heuristics and a method for both creating and conducting concurrent software trace analysis, based on string comparison metrics.

During the previous chapter, we have resorted both to toy programs and the IBM Concurrency benchmark to validate and then evaluate OSCAR both as a noise injection framework and a tool for noise-injection-based testing. The results from the validation phase demonstrated the correctness of OSCAR's noise injection and instrumentation logic.

During these tests, we have verified that programs which had noise injected into their routines by OSCAR consistently generated more interleavings, thus covering more of the program's state space. Furthermore, OSCAR also generated interleavings that were consistently more different from each other. The results show that OSCAR can be reliably used as a noise injection framework for the development of error detection or coverage analysis tools. The vast majority of the IBM Concurrent Benchmark programs, which have latent concurrency errors, saw their errors either manifesting themselves or have their rate of manifestation increase, when subjected to OSCAR's noise injection.

During the run time analysis phase we found that, although the `sleep` noise type is generally the most effective, it is sometimes the most inefficient. The `yield` noise type certainly has a place in time-constrained testing scenarios, such as unit-testing before sending an application to a production environment. Also, although the `yield`'s non-determinism can only be almost completely mitigated by forcing an application to run in a single-core environment, its space state coverage shows equal to slightly better results when running in a multicore environment. This is a big plus to `yield`'s efficiency, since it otherwise required the severe penalty of forfeiting parallelism, which allowed for running a program on additional cores. Another interesting effect of `sleep` is that, it has a clear tendency to mask concurrency errors whose patterns are not being explicitly

covered by the noise injection heuristic. This effect occurs due to the `sleep`'s susceptibility to "distancing" conflict statements or areas in time, lowering their chance of collision.

One major finding, was relative to probabilistic noise injection. We have demonstrated that it can be used reliably to cut the noised program's run times by at least a quarter, without incurring a significant hit to the efficacy of state space exploration. This allows the `sleep`'s major drawback — efficiency — to be considerably mitigated.

In the future, we plan to develop a more accurate representation of interleaving distance. The use of string comparison metrics although valid and with, to our knowledge, most effective when resorting to Jaro, are not ideal, as they only give a measure of syntactic difference between program traces, not taking into account semantic differences such as the distance between statements. Additionally, it is necessary to relate these measurements to program and concurrency coverage, so a tester can have a more precise idea of how much of the state space is being covered.

OSCAR can always be further extended with additional noise placement strategies to build heuristics. These include noising Java concurrency primitives such as `Semaphores`, `Latches`, `Barriers`, `Exchangers` and various other widely-used `Collections`. However, the biggest challenge and perhaps the most relevant, nowadays, would be the noising of `Futures` and any other `Promise`-based concurrency mechanisms, as these asynchronous programming features are currently the standard in web development and, as such, a major source of concurrency errors in popular applications.

However, there are a plethora of possible future improvements to OSCAR. The next section will highlight some of the improvements which we deemed more important, given the insights obtained during the course of our research.

7.2 Future Work

OSCAR was built from the ground up with extendability in mind, stimulating an incremental development approach. This allows researchers and other contributors to the OSCAR open-source project to contribute with many missing features, making OSCAR a more valuable tool for all.

Throughout the research conducted for this project, we found a plethora of desirable features which were not possible to implement given the project's time constraints. However, it makes complete sense to extend OSCAR with these features.

The following sections highlight some of the most notable features with which OSCAR can be extended.

7.2.1 Secondstring — Modified Levenshtein

Cohen et al. [40] developed Secondstring, a novel open-source string comparison Java toolkit, which provided implementations for both well-known and new string comparison metrics. Upon testing the toolkit on a large dataset of names and records, the researchers found that the Jaro metric was very performant. However, they concluded that the Jaro metric may be outperformed

by resorting to a modified Levenshtein distance, more specifically, with a method proposed by Winkler in [197]. It would be interesting to contrast the efficacy and efficiency of this string comparison metric with the Jaro metric on the interleaving information generated by the OSCAR controller's trace.

7.2.2 Strava — MinHashing

Strava [169] is a popular social network for tracking cycling, running and other sports activities which consist in travelling routes, allowing users to share their feats. Arguably the key functionality of the social component of Strava, is the "Activity Grouping" feature, which automatically detects when users are doing a route completely, or only partially, together. To do so, it is necessary to keep track of the users' GPS positions over a given time period, which represent a given route. These routes are then compared with other users', relying on a likeness metric. Initially, the final similarity score between routes was the percentage of time that a given user was within a certain distance of the other users' route. However, as the platform's userbase and activity grew, this method's exponential temporal complexity quickly proved unacceptable [167].

To create a new, more efficient solution, the developers of Strava resorted to MinHashing [27], a technique with an underlying algorithm capable of efficiently estimating the similarity of two sets. MinHashing starts by reducing every sequence of tokens to a set of token subsequences, through a process called *shingling*.

For the string "NoiseInjection", the ω-shingling, when ω is 2, will be:

$$\{No, oi, is, se, eI, In, nj, je, ec, ct, ti, io, on\}$$

From these sets of shingles, it is possible to extract two metrics: resemblance and containment, which are defined in Definition 5. From these formulas, a value between 1 and 0 is derived, which, in the case of resemblance, illustrates how similar two sequences A and B are, and, in the case of containment, how roughly A is contained in B.

Definition 5 (Resemblance and Containment).

Resemblance (Jaccard similarity):

$$R(A,B) = \frac{|S_A \cap S_B|}{|S_A \cup S_B|}$$

Containment:

$$C(A,B) = \frac{|S_A \cap S_B|}{|S_A|}$$

- S_x *- set of all shingles in* X

However, calculating the intersection and union of both A and B explicitly would be temporally complex, especially when given long sets and a larger number of sequences to be compared. As such, MinHashing creates an approximation of similarity. It does so by hashing every shingle of A and B, as a set U, to create an ordered index of hashes. Then, every shingle of A and B

is also hashed, allowing the creation of matrix composed of binary fields. These binary fields represent the presence of a shingle, represented by rows, in a sequence, represented by columns. If one wished to compare a sequence A with more sequences other than just B, it would only be necessary to add additional columns with their respective bitwise representation of shingle hash occurrences.

It is then possible to construct a sequence of the elements of A which occur in B, at their relative position. However, since the bitwise occurrence matrix is computed, it becomes trivial to create permutations of the hashed set of shingles U. The amount of permutations calculated, directly raises the precision of the comparison metric.

The term "Min" in "MinHashing", comes from the mechanism used to create the final matrix. This matrix, with columns representing the same sequences and rows representing each permutation, contains cells which indicate the first (i.e., minimum) value of the index of the permutation array in which a match occurs. Then, a comparison is achieved through a Jaccard similarity, which can be efficiently executed in this shorter produced sequence. If one wishes to compare A and B's similarity, it is only necessary to compute the number of values in A's columns which match with B's column in the comparison matrix, finally dividing this value by the number of rows, yielding a similarity ratio.

Figure 7.1 contains an example comparing 4 permutations of a sequence A with and 4 other sequences. In this example, $S(A,B) = \frac{|\{1\}|}{|\{1,1,1,2\}|} = 0.25$, while $S(A,D) = \frac{|\{1,1,1\}|}{\{|1,1,1,2|\}} = 0.75$.

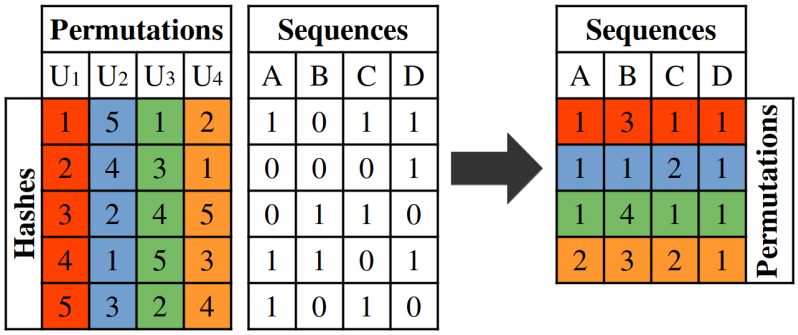


Figure 7.1: An example of the MinHash algorithm.

MinHashing could prove to be an incredibly effective solution for tackling the problem of calculating the likeness of an immense number of long interleavings, presenting itself as an effective solution for interleaving comparison of long and complex software.

More recently, researchers developed SuperMinHash [60], a novel improvement over the original decades-old MinHash algortihm, promising a more efficient implementation which can simultaneously yield more precise likeness estimations.

7.2.3 Two-Phase Noising Mechanism

All the current implementations of noising transformers in OSCAR currently examine statements individually, without analysing the context in which a statement is inserted (i.e., inside a loop, inside an atomic region, etc.). As such, since OSCAR includes heuristics which insert noise before

and after a statement, it is quite normal to see two noise statements one after the other. Although correct, in some instances, this behaviour, in the context of a larger program, will make OSCAR significantly more inefficient. In a best case scenario, where the inserted noise is bounded between zero and some value larger than zero, it becomes statistically possible to reproduce every possible interleaving. However, statistically, it will still be harder to achieve every single possible ordering, because these repeated noising locations will, most likely, insert more noise than intended for this region. An example of this consecutive noise behaviour is present in Listing 4.8 (page 54).

An approach which, at first glance, may seem correct and straightforward, would be to delete one in every two consecutive noise injection statements. The issue with this solution is that noise statements have a specific placement heuristic assigned to them, meaning this approach would make some placement heuristics disappear randomly. This is unacceptable, since the main purpose of OSCAR is to allow the noising of a program with different noise placement and seeding heuristics.

Another worrying instance of noise include locations such as loops, taking the form of `gotos` in bytecode. If a loop block only has a single statement, and this same statement is noised before and after, this creates another instance of consecutive noise. All the cases where this may happen must be carefully studied, such as noising before a method call and directly at the start of the method's routine.

A proposed solution to this issue, would be a two-phased noising process. In the first phase, every noising transformer simply uses the *Soot* API to tag statements with metadata describing where noise will be inserted and what noise placement heuristic that noise is referring to. In a second phase, a more complex analysis is conducted, aggregating all the noise placements into a set. This set then becomes the input of the controller noise statement, meaning this one noising statement can be triggered by multiple noise placement locations. Whenever this statement is reached, during any given SUT run, the controller call will now check if any of the noise placement locations, inside the set that is fed as input is currently active and noising if so.

A method that either resembles or is somewhat equivalent to this proposal, could raise OSCAR's noise injection efficiency, as well as its efficacy at exploring a given program's state space.

7.2.4 Interleaving Replay Mechanism

In Section 3.3.1, ConTest [58] is presented, and its replay mechanism, based on *DejaVu* [37, 110], is briefly described. *DejaVu* (Deterministic Java Replay Utility) is an algorithm which presents itself as a solution to mitigate the characteristic non-determinism of concurrent software. More specifically, the non-determinism encountered when verifying the correctness of some concurrent software implementation through running multiple runs. Without a replay mechanism, it is almost impossible for a tester to reencounter these hidden bugs in its application, leading to dreaded "heisenbugs", since there is no means to reproduce the exact trace that led to the error. As such, two problems arise: a programmer cannot know if he managed to fix a previously encountered bug, and a programmer cannot know if his fix introduced additional bugs. The "cannot" assumes that the program is sufficiently large to have an exponentially larger amount

of possible interleavings, and that the random state space exploration process would require an unacceptable amount of time.

The *DejaVu* algorithm is split into two modes: `record` and `replay`. During the `record` mode, the algorithm records all thread interactions and network traffic that occurs during the interval that the program is running. Whereas, during the `replay` phase, it reproduces the recorded information, enforcing the previous execution into the program, achieving a deterministic replay.

DejaVu captures logical thread scheduling information, since capturing the scheduling information of a physical thread is not always possible. This scheduling information includes every piece of necessary metadata, such as the thread spawn and join order and the order of synchronization operations between threads. The network information is captured via listening to Java's network socket traffic. Additionally, *DejaVu* provides the means to execute in a distributed fashion, allowing several programs to replay the interactions they had between themselves.

The major downside of the original *DejaVu* implementation is that it required a custom JVM implementation to enable the replay capabilities. However, the developers of ConTest managed to implement and equivalent version of the *DejaVu* algorithm, at a bytecode level [58]. Since the OSCAR framework already captures trace information at key locations, it would be very interesting to attempt to empower the OSCAR controller with these replay capabilities. Since the program effectively routinely passes control of the execution to the controller, it may be possible to enforce an order of execution upon arriving at these locations. Unlike *DejaVu* which concerns itself with deterministically replaying every single part of the program, OSCAR could present a more efficient solution, only replaying the necessary logic to guarantee determinism in the locations most likely to affect the scheduling, which is achieved through OSCAR's heuristics. Additionally, adding some control statements to key locations, such as conditional statements, would be feasible and may add additional traceability capabilities.

Initially, it would be interesting to check if a strategy which only concerns itself with replaying the previous noising at every `noise` statement, would be sufficient to achieve, the intended effect of replaying an interleaving.

In [160], researchers make a very significant contribution where many different proposed replay mechanism solutions, including *DejaVu*, are compared. Overall, OSCAR would immensely benefit from a replay mechanism. This would allow a tester to efficiently explore a very large chunk of the state space and deterministically replay any interleaving to fix an error or verify if a previous fix has worked.

7.2.5 I/O Noising

Noise injection is a proven technique for triggering additional interleavings and exploring the state space of programs. However, in the real world, programs do not depend only on the non-determinism originating from the scheduling, but also on non-determinism originating from the program's inputs and outputs. Many modern programs are web services, which in turn need to interact with other network services. If these interactions require a specific order, in order to guarantee correctness, latency and packet loss can cause message races, which may induce

concurrency errors. These errors, although most common in concurrent software written in the message-passing paradigm, are not exclusive to it. In Section 3.3.6, Ninja [172], a noise injection framework for message-passing software was presented. Ninja's algorithm for noising the MPI communications relies on capturing network packets and buffering them inside a virtual queue. Once inside this virtual queue, these packets are then held for a certain interval. The duration of this interval represents the intensity of the network noise.

An approach such as Ninja's, to noise network I/O communications, can be useful to transform OSCAR into a hybrid noise injection solution which can target a wider range of programs, especially modern web-oriented applications. This type of noising may also be paired to a replay mechanism, such as the one presented in Section 7.2.4, can be leveraged to provide determinism for a program tester.

Xue et al. [198] propose a novel replay method called subgroup reproducible replay (SRR), together with MPIWIZ, a prototype to demonstrate its capabilities. The researchers of SRR conjectured that, communication between clustered hosts tends to be asymmetric, with hosts partitioning themselves into disjoint groups made up of the other hosts with whom they communicate most of the time. SRR can infer these groups of hosts and record the order and contents of messages only if its underlying operation is non-deterministic in nature. This is a useful approach, since an operation may occur only between a group of hosts and not the entire universe. Finally, during the replay phase, SRR then proceeds to replay the contents and orderings of the messages exchanged between a particular group.

Until now, OSCAR's focus throughout the document, has been on self-contained applications with a single process. However, many programs, such as distributed applications, require synchronization between different processes. A noising solution such as Ninja, paired with a record and replay mechanism such as SRR, would allow OSCAR to combine shared-memory and message-passing noise injection heuristics in distributed systems and clustered applications.

7.2.6 Coverage Analysis

The purpose of coverage testing is to provide the tester with a notion of test completeness [92] or, in the case of noise injection testing, how much of a program's state space is being explored and validated in the testing process. Since noise injection is a confidence (i.e., best-effort) technique, a tester is almost always sure to explore more of the state space, compared to running the program without any noise. However, it is necessary to have an idea of how many interleavings are possible and relate this number to the number of interleavings which are actually being covered by the noise-injection-based testing.

However, much like inserting noise before and after every statement in the program, guaranteeing coverage of every single part of the state space is an unrealistic and inefficient approach. As such, coverage analysis methods have devised heuristics which identify critical regions where coverage testing is necessary, which, in the case of concurrent software, are the synchronization regions (i.e., synchronization coverage).

Bron et al. [28] defined a coverage analysis method for the Java programming language. The researchers proceeded to leverage *ConTest*'s capabilities to allow for the coverage analysis of a middle-sized multimedia-related project and managed to obtain 100% synchronization coverage in under an hour.

However, ideally, we would wish for a coverage analysis method which does not need to rely on a separate framework, allowing it to be implemented as part of OSCAR. Hong et al. [91] propose a novel coverage method, boasting higher and more efficient coverage capabilities than previously proposed methods. Additionally, the researchers targeted Java concurrent programs with their implementation and open-sourced it. Their proposed thread-scheduling is separated into two-phase: `estimation` and `testing`.

During the `estimation` phase, the algorithm executes the program once to obtain a thread model, consisting of a set of thread executions. In this algorithm, a thread model is defined as a finite set of threads, each with a finite sequence of synchronization actions. From this thread model, the algorithm then estimates the coverage requirements that can be covered by possible thread schedules. The synchronization coverage requirement definition is based on the notion of synchronization pairs, which the authors refer as "SP"s, these are a pair of code locations for which certain conditions hold. These conditions are, for example, that both code locations contain statements that act on a same lock. The `estimation` phase computes and reports this set of SP requirements, but only after passing through a filtering phase which leaves out infeasible pairs. This filtering phase consists of a set of "acceptance conditions" or "AC"s, which are based on the lockset algorithm, and are as follows:

Definition 6 (Acceptance conditions for SP requirements [91]). *Given two lock actions p and q on a lock m:*

If $thread(p) = thread(q)$*:*

1. $q = next(p)$

If $thread(p) \neq thread(q)$*:*

1. $lockset(p, next(p)) \cap lockset(q) = \emptyset$
2. $lockset(p) \cap lockset(prev(q), q) = \emptyset$
3. $q \not\prec p$ *(q does not precede p)*

Where `next(x)` *and* `prev(x)` *refer to the lock actions, executed by the thread that executes x, that access x's operand directly after and before x, respectively.*

The `testing` phase takes the filtered SP requirements, computed in the previous `estimation` phase, and executes the program with a scheduling controller to force the scheduler to achieve more coverage during successive executions. The technique used by the researchers, invokes a method which passes control to the scheduling controller before synchronization events. The controller then computes when it should either pause or resume threads in order to achieve the intended scheduling.

It seems feasible to incorporate an algorithm such as this into OSCAR. The OSCAR logic is already divided into two phases: instrumentation and noising. In the instrumentation phase,

another program transformer can be added to execute the callgraph analysis necessary to compute the SP requirements. Additionally, these requirements can be then saved, in the code or in some file that can later be input into the SUT, for when later executing the noising routine. In the testing phase, since the program already contains a controller that the SUT program passes control onto, through noise statements, it is possible to add the scheduling control capabilities to the OSCAR controllers logic.

7.2.7 Advanced Noise Seeding

Fiedor et al. [63] experimented with seeding techniques which mixed more that one noise type (`sleep` and `yield`), and obtained interesting results. These hybrid noise type seeding techniques managed to yield better results than relying on a single type of noise and manipulating its intensity. These results were obtained via a Read/Write-based noise injection heuristic, with different noise types being used for different types of accesses. The authors reached the conclusion that, while a sleeping thread is blocked, subsequent `yield` calls force the program to quickly switch threads, resulting in more contention, via additional memory accesses.

It would make sense to empower OSCAR with this capability, allowing its noising routine to more than one type of noise and to extend existing and future noise injection heuristics with configuration necessary as to allow a tester to choose which noise type to use in which noise injection locations. The hybrid Read/Write-based noise heuristic presented by the authors, however, would require an additional step of creating noise locations discerning the type of access being done, in contrast to what is currently supported, where access types are not discerned. Another important feature, to be paired with the earlier mentioned mechanisms, would be to allow OSCAR parametrization to accept different noise intensities for different noise types.

Křena et al. [116] suggested a noise injection heuristic which they dub "coverage-based", due to its inherent mechanism's ability to increase concurrency coverage. The heuristic injects noise of any type (e.g., `sleep`, `yield` or `wait`) into a noising location with a given probability, however, this probability is lower the more times this location is reached. Overall, the authors obtained good coverage results from their proposed probabilistic coverage metric.

Currently, in OSCAR, noise is always triggered upon reaching a location. However, it can be triggered with 0 intensity, which is not the same as not triggering at all. It would make sense to empower OSCAR not only with the possibility to add a simple probabilistic noise triggering mechanism, but also to implement the authors' proposal of this probability to be lower, as subsequent accesses are made to a noising location. One of the main perks of probabilistic noise triggering is in performance, as every avoided noising equates to a faster overall program run time. However, tests would have to be conducted, comparing the overall efficacy and efficiency of this approach.

7.2.8 On-The-Fly Healing

Letko et al. [128], from the Brno University of Technology, presented *AtomRace*, a novel algorithm for dynamically detecting (low-level) data races in concurrent programs. This concurrent

software analysis tool was briefly discussed in Section 3.3.3. To detect these data races, *AtomRace* instruments statements before and after accesses to variables or atomic sections, allowing its algorithm to be handed control of the application, similarly to OSCAR. By tracking these accesses, the algorithm can then detect atomicity violations stemming from synchronization.

The authors of *AtomRace*, presented the architecture of their tool as being composed of three components. These components are the following:

Execution Monitor: responsible for monitoring the accesses to variables and atomic sections during the program's execution, keeping track for every access of the thread, variables accessed (in the case atomic sections), location and type of access;

Analysis Engine: the component which implements the actual *AtomRace* algorithm to detect data races from the information obtained by the execution monitor;

Healing Module: in charge of influencing the execution of the Java scheduler to pre-emptly avoid said atomicity violations. The authors warn however, that the implemented techniques only mitigate the probability of the concurrency error to manifest itself, without any assurance that it will actually not do so.

Overall, when combining the *AtomRace* algorithm with the *ConTest* noise injection framework, the authors obtained very good results. This was due to the fact, as pointed out by the authors of *AtomRace*, that, by leveraging noise injection techniques, it becomes possible to highly raise the probability of encountering said data races.

OSCAR's noise injection logic already removes control from the program to inject noise dynamically, making it rather feasible to add an on-the-fly data race tracking and healing mechanism, such as AtomRace. Given OSCAR's modularity, this could be achieved more easily, as a matter of adding the required modules, responsible for this additional logic, and program the instrumentation step to instrument statements into these new locations.

The advantage of using a method such as this is that, by keeping track of shared variables during run time, to detect concurrency errors such as data races and heal them on-the-fly, OSCAR would be able to continue its routine uninterrupted. Meaning, concurrency errors with a higher probability of occurring, would not be allowed to mask others due to halting program execution before these had their opportunity of doing so. Additionally, OSCAR would be enhanced with a built-in error detection tool, instead of relying only on external means, such as I/O parsing (as was done in the Evaluation Section of this dissertation), in order to verify the concurrency of errors.

Bibliography

[1] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory system performance and correctness - MSPC '06*. ACM Press, 2006. DOI: 10.1145/1178597.1178599 (cited on page 4).

[2] G. R. Andrews. *Concurrent programming - principles and practice*. Benjamin-Cummings Publishing Co., Inc., USA, 1st edition, 1991. ISBN: 0805300864 (cited on pages 4, 7, 8, 10, 11).

[3] J. D. N. Arni Einarsson. A survivor's guide to java program analysis with soot, 2008. URL: https://www.brics.dk/SootGuide/ (visited on 09/28/2022) (cited on page 63).

[4] C. Artho, A. Biere, C. Artho, and A. Biere. Combined static and dynamic analysis. *Electronic Notes in Theoretical Computer Science*, 131:3–14, 2005. ISSN: 1571-0661. DOI: https://doi.org/10.1016/j.entcs.2005.01.018. URL: https://www.sciencedirect.com/science/article/pii/S1571066105002537. Proceedings of the First International Workshop on Abstract Interpretation of Object-oriented Languages (AIOOL 2005) (cited on pages 19, 22).

[5] C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003 (cited on pages 1, 2, 12, 13).

[6] C. Artho, K. Havelund, and A. Biere. Using block-local atomicity to detect stale-value concurrency errors. In F. Wang, editor, *Automated Technology for Verification and Analysis*, pages 150–164, Berlin, Heidelberg. Springer Berlin Heidelberg, 2004 (cited on pages 13, 22, 23).

[7] C. Artho, V. Schuppan, A. Biere, P. Eugster, M. Baur, and B. Zweimüller. JNuke: Efficient Dynamic Analysis for Java. In *Computer Aided Verification*, pages 462–465. Springer Berlin Heidelberg, 2004. DOI: 10.1007/978-3-540-27813-9_37 (cited on page 22).

[8] S. A. Asadollah, D. Sundmark, S. Eldh, and H. Hansson. Concurrency bugs in open source software: a case study. *Journal of Internet Services and Applications*, 8(1), Apr. 2017. DOI: 10.1186/s13174-017-0055-2 (cited on page 17).

[9] H. Attiya and J. L. Welch. Sequential Consistency versus Linearizability. *ACM Trans. Comput. Syst.*, 12(2):91–122, May 1994. ISSN: 0734-2071. DOI: 10.1145/176575.176576. URL: https://doi.org/10.1145/176575.176576 (cited on page 7).

[10] D. A. B. *The Little Book of Semaphores*. A. B. Downey, editor. Green Tea Press, second edition, 2016. URL: https://greenteapress.com/wp/semaphores/ (cited on page 9).

[11] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018 (cited on page 20).

[12] T. Ball. The concept of dynamic analysis. *SIGSOFT Softw. Eng. Notes*, 24(6):216–234, Oct. 1999. ISSN: 0163-5948. DOI: 10.1145/318774.318944. URL: https://doi.org/10.1145/318774.318944 (cited on page 19).

[13] C. Baquero and N. Preguiça. Why Logical Clocks Are Easy. *Commun. ACM*, 59(4):43–47, Mar. 2016. ISSN: 0001-0782. DOI: 10.1145/2890782. URL: https://doi.org/10.1145/2890782 (cited on page 8).

[14] I. D. Baxter. Dms: practical code generation and enhancement by program transformation. In *Workshop on Generative Programming*, pages 19–20. Citeseer, 2002 (cited on page 35).

[15] S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Lecture Notes in Computer Science*, pages 208–223. Springer Berlin Heidelberg, 2006. DOI: 10.1007/11678779_15 (cited on page 23).

[16] A. J. Bernstein. Analysis of Programs for Parallel Processing. *IEEE Transactions on Electronic Computers*, EC-15:757–763, 1966. ISSN: 0367-7508. DOI: 10.1109/pgec.1966.264565. URL: http://doi.org/10.1109/pgec.1966.264565 (cited on page 12).

[17] W. Binder, J. Hulaas, and P. Moret. Advanced java bytecode instrumentation. In *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java*, PPPJ '07, pages 135–144, Lisboa, Portugal. Association for Computing Machinery, 2007. ISBN: 9781595936721. DOI: 10.1145/1294325.1294344. URL: https://doi.org/10.1145/1294325.1294344 (cited on pages 33, 36, 37, 62).

[18] S. Blackshear, N. Gorogiannis, P. W. O'Hearn, and I. Sergey. RacerD: compositional static race detection. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):1–28, Oct. 2018. DOI: 10.1145/3276514 (cited on page 21).

[19] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Reliable Software Technologies Ada-Europe 2000*, pages 225–237. Springer Berlin Heidelberg, 2000. DOI: 10.1007/10722060_21 (cited on page 23).

[20] J. S. Bradbury and K. Jalbert. Defining a catalog of programming anti-patterns for concurrent java. In *Proc. of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)*, pages 6–11, 2009 (cited on page 80).

[21] J. S. Bradbury, I. Segall, E. Farchi, K. Jalbert, and D. Kelk. Using combinatorial benchmark construction to improve the assessment of concurrency bug detection tools. In *Proceedings of the 2012 Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD 2012, pages 25–35, Minneapolis, MN, USA. Association for Computing Machinery, 2012. ISBN: 9781450314565. DOI: 10.1145/2338967.2336812. URL: https://doi.org/10.1145/2338967.2336812 (cited on pages 33, 80).

[22] G. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25:167–198, Sept. 2004. DOI: 10.1023/B:FORM.0000040027.28662.a4 (cited on page 33).

[23] C. Breshears. *The Art of Concurrency: A Thread Monkey's Guide to Writing Parallel Applications*. O'Reilly Media, Inc., 2009. ISBN: 0596521537 (cited on pages 4–6, 12, 15, 16).

[24] E. Brewer. CAP twelve years later: How the "rules" have changed. *Computer*, 45(2):23–29, Feb. 2012. DOI: 10.1109/mc.2012.37 (cited on page 7).

[25] E. A. Brewer. Towards robust distributed systems (abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing - PODC '00*. ACM Press, 2000. DOI: 10.1145/343477.343502 (cited on page 7).

[26] H. Brinch and P. Hansen. *Operating System Principles*. Prentice-Hall series in automatic computation. Prentice-Hall, 1973. ISBN: 9780136378433. URL: https://books.google.com/books?id=lNEmAAAAMAAJ (cited on page 11).

[27] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of SEQUENCES 1997 (Cat. No.97TB100171)*, pages 21–29, 1997. DOI: 10.1109/SEQUEN.1997.666900 (cited on page 106).

[28] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 206–212, Chicago, IL, USA. Association for Computing Machinery, 2005. ISBN: 1595930809. DOI: 10.1145/1065944.1065972. URL: https://doi.org/10.1145/1065944.1065972 (cited on page 111).

[29] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: a code manipulation tool to implement adaptable systems. *Adaptable and extensible component systems*, 30(19), 2002 (cited on page 35).

[30] B. Buck and J. K. Hollingsworth. An api for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, 2000 (cited on page 38).

[31] M. Burrows and K. R. M. Leino. Finding stale-value errors in concurrent programs. *Concurrency and Computation: Practice and Experience*, 16(12):1161–1172, Aug. 2004. DOI: 10.1002/cpe.866 (cited on pages 15, 22).

[32] V. Buterin. A next-generation smart contract and decentralized application platform. *white paper*, 2015. URL: https://blockchainlab.com/pdf/Ethereum_white_paper-a_next_generation_smart_contract_and_decentralized_application_platform-vitalik-buterin.pdf (visited on 09/28/2022) (cited on page 7).

[33] R. O. Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '03*. ACM Press, 2003. DOI: 10.1145/781498.781528 (cited on page 21).

[34] Q. Chen, L. Wang, Z. Yang, and S. D. Stoller. HAVE: Detecting Atomicity Violations via Integrated Dynamic and Static Analysis. In *Fundamental Approaches to Software Engineering*, pages 425–439. Springer Berlin Heidelberg, 2009. DOI: 10.1007/978-3-642-00593-0_30 (cited on page 21).

[35] B. Chess and G. McGraw. Static analysis for security. *IEEE security & privacy*, 2(6):76–79, 2004 (cited on page 19).

[36] L. Chew and D. Lie. Kivati. In *Proceedings of the 5th European conference on Computer systems - EuroSys '10*. ACM Press, 2010. DOI: 10.1145/1755913.1755945 (cited on page 21).

[37] J.-D. Choi and H. Srinivasan. Deterministic Replay of Java Multithreaded Applications. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT '98*. ACM Press, 1998. DOI: 10.1145/281035.281041 (cited on pages 39, 108).

[38] E. M. Clarke Jr, O. Grumberg, D. Kroening, D. Peled, and H. Veith. *Model checking. The Cyber-Physical Systems Series*. MIT press, 2nd edition, 2018. ISBN: 9780262038836 (cited on page 23).

[39] G. A. Cohen, J. S. Chase, and D. L. Kaminsky. Automatic program transformation with JOIE. In *1998 USENIX Annual Technical Conference (USENIX ATC 98)*, New Orleans, LA. USENIX Association, June 1998. URL: https://www.usenix.org/conference/1998-usenix-annual-technical-conference/automatic-program-transformation-joie (cited on page 35).

[40] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string metrics for matching names and records. *Proc of the KDD Workshop on Data Cleaning and Object Consolidation*, Oct. 2003 (cited on pages 58, 105).

[41] J. Cordy, C. Halpern, and E. Promislow. TXL: a rapid prototyping system for programming language dialects. In *Proceedings. 1988 International Conference on Computer Languages*, pages 280–285, 1988. DOI: 10.1109/ICCL.1988.13075 (cited on page 34).

[42] J. R. Cordy, T. R. Dean, A. J. Malton, and K. A. Schneider. Source transformation in software engineering using the TXL transformation system. *Information and Software Technology*, 44(13):827–837, Oct. 2002. DOI: 10.1016/s0950-5849(02)00104-0 (cited on page 34).

[43] J. R. Cordy. TXL - A Language for Programming Language Tools and Applications. *Electronic Notes in Theoretical Computer Science*, 110:3–31, Dec. 2004. DOI: 10.1016/j.entcs.2004.11.006 (cited on page 34).

[44] F. Cristian. Probabilistic clock synchronization. *Distributed computing*, 3(3):146–158, 1989 (cited on page 8).

[45] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed systems*, 10(6):642–657, 1999 (cited on page 8).

[46] L. Dagum and R. Menon. Openmp: an industry standard api for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998. DOI: 10.1109/99.660313 (cited on page 36).

[47] M. Dahm. Byte Code Engineering with the BCEL API. Technical Report B-17-98. Technical report, Freie Universität Berlin Institut für Informatik, 2001 (cited on pages 35, 37).

[48] F. J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, Mar. 1964. ISSN: 0001-0782. DOI: 10.1145/363958.363994 (cited on pages 57, 74).

[49] F. M. David, J. C. Carlyle, and R. H. Campbell. Context Switch Overheads for Linux on ARM Platforms. In *Proceedings of the 2007 Workshop on Experimental Computer Science*, ExpCS '07, New York, NY, USA. Association for Computing Machinery, 2007. ISBN: 9781595937513. DOI: 10.1145/1281700.1281703. URL: https://doi.org/10.1145/1281700.1281703 (cited on page 11).

[50] R. C. de Amorim and M. Zampieri. Effective spell checking methods using clustering algorithms. In *Proceedings of the International Conference Recent Advances in Natural Language Processing RANLP 2013*, pages 172–178, Hissar, Bulgaria. INCOMA Ltd. Shoumen, BULGARIA, Sept. 2013. URL: https://aclanthology.org/R13-1023 (cited on page 56).

[51] C. Delporte-Gallet, H. Fauconnier, and R. Guerraoui. Shared Memory vs Message Passing. Technical report, EPFL - DCL (Distributed Computing Laboratory), 2003 (cited on page 5).

[52] F. de Luna. Github: oscar noise injection framework, 2022. URL: https://github.com/filipedeluna/oscar (cited on page 61).

[53] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. In volume 29, number 7, pages 577–603. Wiley, June 1999. DOI: 10.1002/(SICI)1097-024X(199906)29:7<577::AID-SPE246>3.0.CO;2-V (cited on page 23).

[54] R. J. Dias, V. Pessanha, and J. M. Lourenço. Precise Detection of Atomicity Violations. In *Hardware and Software: Verification and Testing*, pages 8–23. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-39611-3_8 (cited on pages 9, 12, 13, 22).

[55] E. W. Dijkstra. Solution of a Problem in Concurrent Programming Control. *Commun. ACM*, 8(9):569, Sept. 1965. ISSN: 0001-0782. DOI: 10.1145/365559.365617. URL: https://doi.org/10.1145/365559.365617 (cited on page 9).

[56] L. K. Dillon. Using symbolic execution for verification of ada tasking programs. *ACM Trans. Program. Lang. Syst.*, 12(4):643–669, Oct. 1990. ISSN: 0164-0925. DOI: 10.1145/88616.96551. URL: https://doi.org/10.1145/88616.96551 (cited on page 20).

[57] J. S. 8. Documentation. Java Thread Primitive Deprecation. Oracle. 2021. URL: https://docs.oracle.com/javase/8/docs/technotes/guides/concurrency/threadPrimitiveDeprecation.html (cited on page 28).

[58] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded java program test generation. *IBM Systems Journal*, 41(1):111–125, 2002. DOI: 10.1147/sj.411.0111 (cited on pages 2, 23, 26–29, 31–33, 37, 38, 45, 49, 61, 69, 79, 86, 108, 109).

[59] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks. *Communications of the ACM*, 53(11):85–92, Nov. 2010. DOI: 10.1145/1839676.1839698 (cited on page 40).

[60] O. Ertl. Superminhash - a new minwise hashing algorithm for jaccard similarity estimation, 2017. DOI: 10.48550/ARXIV.1706.05698 (cited on page 107).

[61] B. Evans. Behind the scenes: how do lambda expressions really work in java? J. Magazine, editor. Sept. 28, 2018. URL: https://blogs.oracle.com/javamagazine/post/behind-the-scenes-how-do-lambda-expressions-really-work-in-java (visited on 04/25/2022) (cited on page 70).

[62] Y. Eytani, E. Farchi, and Y. Ben-Asher. Heuristics for finding concurrent bugs. In *Proceedings International Parallel and Distributed Processing Symposium*. IEEE Comput. Soc, 2003. DOI: 10.1109/ipdps.2003.1213514 (cited on pages 31, 32).

[63] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in noise-based testing of concurrent software. *Software Testing, Verification and Reliability*, 25(3):272–309, 2015. DOI: https://doi.org/10.1002/stvr.1546. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1546 (cited on pages 2, 19, 23, 26–29, 31, 33, 39, 95, 112).

[64] J. Fiedor and T. Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In S. Qadeer and S. Tasiran, editors, *Runtime Verification*, pages 35–41, Berlin, Heidelberg. Springer Berlin Heidelberg, 2013. ISBN: 978-3-642-35632-2 (cited on pages 1, 19, 36, 39).

[65] J. Fiedor and T. Vojnar. Noise-based testing and analysis of multi-threaded c/c++ programs on the binary level. *2012 10th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging, PADTAD 2012 - Proceedings*, July 2012. DOI: 10.1145/2338967.2336813 (cited on pages 29, 31, 36, 69).

[66] C. Flanagan and S. N. Freund. Atomizer: a dynamic atomicity checker for multithreaded programs. *ACM SIGPLAN Notices*, 39(1):256–267, 2004 (cited on pages 21, 40).

[67] C. Flanagan and S. N. Freund. FastTrack. *ACM SIGPLAN Notices*, 44(6):121–133, May 2009. DOI: 10.1145/1543135.1542490 (cited on page 40).

[68] C. Flanagan and S. N. Freund. The RoadRunner Dynamic Analysis Framework for Concurrent Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '10, pages 1–8, New York, NY, USA. Association for Computing Machinery, 2010. ISBN: 9781450300827. DOI: 10.1145/1806672.1806674 (cited on pages 1, 40, 61).

[69] C. Flanagan and S. Qadeer. A Type and Effect System for Atomicity. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 338–349, New York, NY, USA. Association for Computing Machinery, 2003. ISBN: 1581136625. DOI: 10.1145/781131.781169. URL: https://doi.org/10.1145/781131.781169 (cited on page 9).

[70] C. Flanagan and J. B. Saxe. Avoiding exponential explosion: generating compact verification conditions. *SIGPLAN Not.*, 36(3):193–205, Jan. 2001. ISSN: 0362-1340. DOI: 10.1145/373243.360220 (cited on page 20).

[71] J. Gait. A probe effect in concurrent programs. *Software: Practice and Experience*, 16(3):225–233, 1986. DOI: https://doi.org/10.1002/spe.4380160304. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.4380160304 (cited on pages 2, 19, 26).

[72] L. F. R. Garcia. *Automatic fault localization in concurrent programs using noising and search strategies.* Master's thesis, University of Ontario Institute of Technology, 2020 (cited on pages 31, 32, 34, 41, 44, 46).

[73] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java concurrency in practice.* Pearson Education, 2006. ISBN: 9780321349606 (cited on pages 26, 44, 48).

[74] J. Gray. Why do computers stop and what can be done about it? In *Symposium on reliability in distributed software and database systems*, pages 3–12. Los Angeles, CA, USA, 1986 (cited on page 26).

[75] R. Gusella and S. Zatti. The accuracy of the clock synchronization achieved by TEMPO in Berkeley UNIX 4.3 BSD. *IEEE transactions on Software Engineering*, 15(7):847–853, 1989 (cited on page 8).

[76] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic detection of atomic-set-serializability violations. In *Proceedings of the 13th international conference on Software engineering - ICSE '08.* ACM Press, 2008. DOI: 10.1145/1368088.1368120 (cited on page 22).

[77] R. W. Hamming. Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160, 1950. DOI: 10.1002/j.1538-7305.1950.tb00463.x (cited on pages 57, 74).

[78] H. Hansen, W. Penczek, and A. Valmari. Stuttering-insensitive automata for on-the-fly detection of livelock properties. *Electronic Notes in Theoretical Computer Science*, 66(2):178–193, 2002 (cited on page 23).

[79] T. Hansen, P. Schachte, and H. Søndergaard. State joining and splitting for the symbolic execution of binaries. In S. Bensalem and D. A. Peled, editors, *Runtime Verification*, pages 76–92, Berlin, Heidelberg. Springer Berlin Heidelberg, 2009. ISBN: 978-3-642-04694-0 (cited on page 20).

[80] K. Havelund, S. Stoller, and S. Ur. Benchmark and framework for encouraging research on multi-threaded testing tools. In *Proceedings International Parallel and Distributed Processing Symposium*, page 286, 2003. DOI: 10.1109/IPDPS.2003.1213510 (cited on page 79).

[81] L. Hendren. Uses of the soot framework. M. University, editor, 2018. URL: https://www.sable.mcgill.ca/~hendren/sootusers/ (visited on 09/28/2022) (cited on page 35).

[82] M. Herlihy, N. Shavit, V. Luchangco, and M. Spear. *The Art of Multiprocessor Programming*. Morgan Kaufmann, Boston, second edition, 2020. ISBN: 978-0-12-415950-1. DOI: https://doi.org/10.1016/B978-0-12-415950-1.00009-4 (cited on pages 1, 5, 6, 11, 17).

[83] A. Ho, S. Smith, and S. Hand. On deadlock, livelock, and forward progress. Technical report UCAM-CL-TR-633, University of Cambridge, Computer Laboratory, May 2005. DOI: 10.48456/tr-633 (cited on page 16).

[84] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969. ISSN: 0001-0782. DOI: 10.1145/363235.363259 (cited on page 19).

[85] C. A. R. Hoare. Monitors: An Operating System Structuring Concept. *Commun. ACM*, 17(10):549–557, Oct. 1974. ISSN: 0001-0782. DOI: 10.1145/355620.361161 (cited on page 11).

[86] V. Hodge and J. Austin. A comparison of standard spell checking algorithms and a novel binary neural approach. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1073–1081, Sept. 2003. DOI: 10.1109/TKDE.2003.1232265 (cited on page 57).

[87] J. K. Hollingsworth, O. Niam, B. P. Miller, Z. Xu, M. J. Gonçalves, and L. Zheng. Mdl: a language and compiler for dynamic program instrumentation. In *Proceedings 1997 International Conference on Parallel Architectures and Compilation Techniques*, pages 201–212. IEEE, 1997 (cited on page 38).

[88] R. C. Holt. Some Deadlock Properties of Computer Systems. *ACM Computing Surveys*, 4(3):179–196, Sept. 1972. DOI: 10.1145/356603.356607 (cited on page 15).

[89] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997. DOI: 10.1109/32.588521 (cited on page 23).

[90] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., USA, 1990. ISBN: 0135399254 (cited on page 23).

[91] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, ISSTA 2012, pages 210–220, Minneapolis, MN, USA. Association for Computing Machinery, 2012. ISBN: 9781450314541. DOI: 10.1145/2338965.2336779 (cited on page 111).

[92] J. Horgan, S. London, and M. Lyu. Achieving software quality with testing coverage measures. *Computer*, 27(9):60–69, 1994. DOI: 10.1109/2.312032 (cited on page 110).

[93] V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Search Based Software Engineering*, pages 152–167. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-33119-0_12 (cited on pages 29, 32, 41, 88).

[94] W. Hseush and G. E. Kaiser. Modeling concurrency in parallel debugging. In *Proceedings of the second ACM SIGPLAN symposium on Principles & practice of parallel programming - PPOPP '90*. ACM Press, 1990. DOI: 10.1145/99163.99166 (cited on page 24).

[95] IEEE. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, 1990. DOI: 10.1109/IEEESTD.1990.101064 (cited on pages 18, 19).

[96] N. Jalbert and K. Sen. A trace simplification technique for effective debugging of concurrent programs. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 57–66, Santa Fe, New Mexico, USA. Association for Computing Machinery, 2010. ISBN: 9781605587912. DOI: 10.1145/1882291.1882302 (cited on page 54).

[97] M. Jaro. Advances in record-linkage methodology as applied to matching the 1985 census of tampa, florida. *Journal of The American Statistical Association - J AMER STATIST ASSN*, 84:414–420, June 1989. DOI: 10.1080/01621459.1989.10478785 (cited on pages 58, 59, 74).

[98] A. R. Jim Gray. *Transaction Processing: Concepts and Techniques*. The Morgan Kaufmann Series in Data Management Systems. Morgan Kaufmann, first edition, 1992. ISBN: 9781558601901 (cited on page 10).

[99] M. Jin. Ida2obj: static binary instrumentation on steroids. 2021. URL: https://conference.hitb.org/hitbsecconf2021sin/materials/D1T2%20-%20IDA2Obj%20-%20Static%20Binary%20Instrumentation%20On%20Steroids%20-%20Mickey%20Jin.pdf. HITB SECCONF SIN 2021 (cited on page 36).

[100] K. A. John C. Mitchell. *Concepts in Programming Languages*. Cambridge University Press, first edition, 2001. ISBN: 9780521780988 (cited on page 18).

[101] H. Jula, D. Tralamazza, C. Zamfir, and G. Candea. Deadlock immunity: enabling systems to defend against deadlocks. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 295–308, San Diego, California. USENIX Association, 2008 (cited on pages 5, 11).

[102] M. Kadin and S. Reda. Frequency and voltage planning for multi-core processors under thermal constraints. In *2008 IEEE International Conference on Computer Design*, pages 463–470. IEEE, Oct. 2008. DOI: 10.1109/iccd.2008.4751902 (cited on page 1).

[103] V. Kahlon, S. Sankaranarayanan, and A. Gupta. Static analysis for concurrent programs with applications to data race detection. *Int. J. Softw. Tools Technol. Transf.*, 15(4):321–336, Aug. 2013. ISSN: 1433-2779. DOI: 10.1007/s10009-013-0274-1 (cited on page 19).

[104] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and Accurate Static Data-Race Detection for Concurrent Programs. In *Computer Aided Verification*, pages 226–239. Springer Berlin Heidelberg, 2007. DOI: 10.1007/978-3-540-73368-3_26 (cited on pages 20, 21, 23).

[105] S. R. Kay A. Robbins. *UNIX systems programming: communication, concurrency and threads*. Prentice Hall, second edition, 2003. ISBN: 9780130424112 (cited on page 11).

[106] L. S. Keller. Operating Systems. In R. A. Meyers, editor, *Encyclopedia of Physical Science and Technology (Third Edition)*, pages 169–191. Academic Press, New York, third edition edition, 2003. ISBN: 978-0-12-227410-7. DOI: https://doi.org/10.1016/B0-12-227410-5/00851-6 (cited on page 16).

[107] K. Kim, T. Yavuz-Kahveci, and B. A. Sanders. Jrf-e: using model checking to give advice on eliminating memory model-related bugs. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 215–224, Antwerp, Belgium. Association for Computing Machinery, 2010. ISBN: 9781450301169. DOI: 10.1145/1858996.1859042 (cited on page 87).

[108] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976 (cited on page 20).

[109] J. C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, Apr. 1975. ISSN: 0362-1340. DOI: 10.1145/390016.808444 (cited on page 20).

[110] R. Konuru, H. Srinivasan, and J.-D. Choi. Deterministic replay of distributed java applications. In *Proceedings 14th International Parallel and Distributed Processing Symposium. IPDPS 2000*, pages 219–227, 2000. DOI: 10.1109/IPDPS.2000.845988 (cited on page 108).

[111] H. Kopetz, G. Grünsteidl, and J. Reisinger. Fault-tolerant membership service in a synchronous distributed real-time system. In *Dependable Computing for Critical Applications*, pages 411–429. Springer, 1991 (cited on page 8).

[112] H. Kopetz and W. Ochsenreiter. Clock synchronization in distributed real-time systems. *IEEE Transactions on Computers*, 100(8):933–940, 1987 (cited on page 8).

[113] E. Koskinen and M. Herlihy. Dreadlocks. In *Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures - SPAA '08*. ACM Press, 2008. DOI: 10.1145/1378533.1378585 (cited on page 23).

[114] B. Krena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing data races on-the-fly. In *proceedings of the 2007 ACM Workshop on Parallel and Distributed Systems: Testing and Debugging*, pages 54–64, 2007 (cited on pages 12, 32).

[115] B. Křena, Z. Letko, and T. Vojnar. Coverage metrics for saturation-based and search-based testing of concurrent software. In *Proceedings of the Second International Conference on Runtime Verification*, RV'11, pages 177–192, San Francisco, CA. Springer-Verlag, 2011. ISBN: 9783642298592. DOI: 10.1007/978-3-642-29860-8_14 (cited on page 55).

[116] B. Křena, Z. Letko, and T. Vojnar. Noise injection heuristics for concurrency testing. In Z. Kotásek, J. Bouda, I. Černá, L. Sekanina, T. Vojnar, and D. Antoš, editors, *Mathematical and Engineering Methods in Computer Science*, pages 123–135, Berlin, Heidelberg. Springer Berlin Heidelberg, 2012. ISBN: 978-3-642-25929-6 (cited on pages 2, 29, 112).

[117] A. Laarman and D. Faragó. Improved on-the-Fly Livelock Detection. In *Lecture Notes in Computer Science*, pages 32–47. Springer Berlin Heidelberg, 2013. DOI: 10.1007/978-3-642-38088-4_3 (cited on page 23).

[118] P. Lam, E. Bodden, O. Lhoták, and L. Hendren. The soot framework for java program analysis: a retrospective. In *CETUS 2011*, Oct. 2011 (cited on page 35).

[119] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, 1977. DOI: 10.1109/TSE.1977.229904 (cited on page 8).

[120] L. Lamport. A New Solution of Dijkstra's Concurrent Programming Problem. *Commun. ACM*, 17(8):453–455, Aug. 1974. ISSN: 0001-0782. DOI: 10.1145/361082.361093. URL: https://doi.org/10.1145/361082.361093 (cited on page 9).

[121] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM*, 21(7):558–565, July 1978. ISSN: 0001-0782. DOI: 10.1145/359545.359563 (cited on pages 5, 8).

[122] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely. Pebil: Efficient static binary instrumentation for linux. In *2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS)*, pages 175–183. IEEE, 2010 (cited on pages 33, 35, 36).

[123] E. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006. DOI: 10.1109/mc.2006.180 (cited on page 4).

[124] H. Lee and B. Zorn. Bit: a tool for instrumenting java bytecodes. *USENIX Symposium on Internet Technologies and Systems*, Dec. 1997 (cited on page 37).

[125] J. Leitao, J. Pereira, and L. Rodrigues. HyParView: A Membership Protocol for Reliable Gossip-Based Broadcast. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*, pages 419–429, 2007. DOI: 10.1109/DSN.2007.56 (cited on page 8).

[126] Z. Letko. Analysis and testing of concurrent programs. *Information Sciences & Technologies: Bulletin of the ACM Slovakia*, 5(3), 2013 (cited on page 29).

[127] Z. Letko. *Analysis and Sophisticated Testing of Concurrent Programs*. PhD thesis, Brno University of Technology, Mar. 2010 (cited on page 32).

[128] Z. Letko, T. Vojnar, and B. Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging*, PADTAD '08, New York, NY, USA. Association for Computing Machinery, 2008. ISBN: 9781605580524. DOI: 10.1145/1390841.1390848 (cited on pages 1, 12, 26, 27, 40, 112).

[129] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10 of number 8, pages 707–710. Soviet Union, 1966 (cited on pages 57, 74).

[130] T. Li, A. Lebeck, and D. Sorin. Spin detection hardware for improved management of multithreaded systems. *IEEE Transactions on Parallel and Distributed Systems*, 17(6):508–521, 2006. DOI: 10.1109/TPDS.2006.78 (cited on page 23).

[131] T. Li, C. S. Ellis, A. R. Lebeck, and D. J. Sorin. Pulse: A Dynamic Deadlock Detection Mechanism Using Speculative Execution. In *USENIX Annual Technical Conference, General Track*, volume 44, 2005 (cited on page 23).

[132] B. Liskov and R. Ladin. Highly available distributed services and fault-tolerant distributed garbage collection. In *Proceedings of the fifth annual ACM symposium on Principles of distributed computing*, pages 29–39, 1986 (cited on page 8).

[133] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/master/template.pdf (cited on page ii).

[134] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. *SIGOPS Oper. Syst. Rev.*, 42(2):329–339, Mar. 2008. ISSN: 0163-5980. DOI: 10.1145/1353535.1346323. URL: https://doi.org/10.1145/1353535.1346323 (cited on pages 13–15).

[135] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO. *ACM SIGOPS Operating Systems Review*, 40(5):37–48, Oct. 2006. DOI: 10.1145/1168917.1168864 (cited on page 21).

[136] B. Lucia, J. Devietti, K. Strauss, and L. Ceze. Atom-Aid: Detecting and Surviving Atomicity Violations. In *2008 International Symposium on Computer Architecture*, pages 277–288, 2008. DOI: 10.1109/ISCA.2008.4 (cited on page 14).

[137] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. *Acm sigplan notices*, 40(6):190–200, 2005 (cited on page 62).

[138] H. Ma, S. R. Diersen, L. Wang, C. Liao, D. Quinlan, and Z. Yang. Symbolic Analysis of Concurrency Errors in OpenMP Programs. In *2013 42nd International Conference on Parallel Processing*, pages 510–516, 2013. DOI: 10.1109/ICPP.2013.63 (cited on pages 21, 23).

[139] A. S. T. Maarten van Steen. *Distributed Systems*. CreateSpace IPP, third edition, 2017. ISBN: 9781543057386 (cited on pages 6–8, 15).

[140] L. Marek, Y. Zheng, D. Ansaloni, A. Sarimbekov, W. Binder, P. Tůma, and Z. Qi. Java bytecode instrumentation made easy: the DiSL framework for dynamic program analysis. In *Programming Languages and Systems*, pages 256–263. Springer Berlin Heidelberg, 2012. DOI: 10.1007/978-3-642-35182-2_18 (cited on page 37).

[141] D. Marino, M. Musuvathi, and S. Narayanasamy. LiteRace. In *Proceedings of the 2009 ACM SIGPLAN conference on Programming language design and implementation - PLDI '09*. ACM Press, 2009. DOI: 10.1145/1542476.1542491 (cited on page 21).

[142] S. P. Masticola and B. G. Ryder. A model of Ada programs for static deadlock detection in polynomial times. *ACM SIGPLAN Notices*, 26(12):97–107, Dec. 1991. DOI: 10.1145/127695.122768 (cited on page 23).

[143] P. Maymounkov and D. Mazières. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In *Peer-to-Peer Systems*, pages 53–65. Springer Berlin Heidelberg, 2002. DOI: 10.1007/3-540-45748-8_5 (cited on page 7).

[144] M. McCool, A. D. Robison, and J. Reinders. *Structured Parallel Programming*. M. McCool, A. D. Robison, and J. Reinders, editors. Morgan Kaufmann, Boston, 2012, pages 1–38. ISBN: 978-0-12-415993-8. DOI: https://doi.org/10.1016/B978-0-12-415993-8.00001-3 (cited on pages 1, 11).

[145] C. E. McDowell and D. P. Helmbold. Debugging concurrent programs. *ACM Comput. Surv.*, 21(4):593–622, Dec. 1989. ISSN: 0360-0300. DOI: 10.1145/76894.76897 (cited on page 19).

[146] D. Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014 (cited on page 41).

[147] D. Mills. Internet time synchronization: the network time protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, 1991. DOI: 10.1109/26.103043 (cited on page 8).

[148] G. E. Moore. Cramming more components onto integrated circuits. *Electronics, volume 38, number 8, April 19, 1965*, 11(3):33–35, Sept. 2006. DOI: 10.1109/n-ssc.2006.4785860 (cited on page 1).

[149] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective static deadlock detection. In *2009 IEEE 31st International Conference on Software Engineering*. IEEE, 2009. DOI: 10.1109/icse.2009.5070538 (cited on page 23).

[150] S. Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. *Cryptography Mailing list at https://metzdowd.com*, Mar. 2009 (cited on page 7).

[151] N. Nethercote. Dynamic binary analysis and instrumentation. Technical report, University of Cambridge, Computer Laboratory, 2004. URL: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-606.pdf (visited on 09/28/2022) (cited on pages 18, 19, 33).

[152] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003 (cited on page 38).

[153] N. Nethercote and J. Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. *ACM Sigplan notices*, 42(6):89–100, 2007 (cited on page 38).

[154] R. H. B. Netzer, T. W. Brennan, and S. K. Damodaran-Kamal. Debugging race conditions in message-passing programs. In *Proceedings of the SIGMETRICS symposium on Parallel and distributed tools - SPDT '96*. ACM Press, 1996. DOI: 10.1145/238020.238033 (cited on page 42).

[155] Oracle. Reentrantlock. 2020. URL: https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/locks/ReentrantLock.html (visited on 04/27/2022) (cited on page 48).

[156] C.-S. Park and K. Sen. Randomized active atomicity violation detection in concurrent programs. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering - SIGSOFT '08/FSE-16*. ACM Press, 2008. DOI: 10.1145/1453101.1453121 (cited on page 21).

[157] S. Park, S. Lu, and Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 25–36, New York, NY, USA. Association for Computing Machinery, 2009. ISBN: 9781605584065. DOI: 10.1145/1508244.1508249 (cited on pages 14, 21).

[158] O. L. Patrick Lam Feng Qian. Soot phase options. 2022. URL: https://www.sable.mcgill.ca/soot/tutorial/phase/phase.html (visited on 03/14/2022) (cited on pages 64, 65).

[159] G. L. Peterson. Myths About the Mutual Exclusion Problem. *Inf. Process. Lett.*, 12:115–116, 1981 (cited on page 17).

[160] G. A. Pokam, C. L. Pereira, K. Danne, L. Yang, S. T. King, and J. Torrellas. Hardware and software approaches for deterministic multi-processor replay of concurrent programs. In *Intel Technology Journal*, 2009 (cited on page 109).

[161] P. Pratikakis, J. S. Foster, and M. Hicks. LOCKSMITH. *ACM Transactions on Programming Languages and Systems*, 33(1):1–55, Jan. 2011. DOI: 10.1145/1889997.1890000 (cited on page 21).

[162] C. R. Prause, R. Reiners, and S. Dencheva. Empirical study of tool support in highly distributed research projects. In *2010 5th IEEE International Conference on Global Software Engineering*, pages 23–32, 2010. DOI: 10.1109/ICGSE.2010.13 (cited on page 18).

[163] W. Pugh and N. Ayewah. Unit testing concurrent software. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 513–516, 2007 (cited on pages 1, 2).

[164] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2012. ISBN: 3642320260 (cited on pages 4, 8–11, 17).

[165] M. Raynal. *Distributed Algorithms for Message-Passing Systems*. Springer Publishing Company, Incorporated, 2013. ISBN: 3642381227 (cited on page 7).

[166] V. J. Reddi, A. Settle, D. A. Connors, and R. S. Cohn. PIN: a binary instrumentation tool for computer architecture research and education. In *WCAE 2004*, 2004 (cited on pages 22, 37, 39).

[167] D. Robb. Activity grouping: the heart of a social network for athletes, Dec. 2018. URL: https://medium.com/strava-engineering/activity-grouping-the-heart-of-a-social-network-for-athletes-865751f7dca (cited on page 106).

[168] S. Robbins. Starving Philosophers: Experimentation with Monitor Synchronization. *SIGCSE Bull.*, 33(1):317–321, Feb. 2001. ISSN: 0097-8418. DOI: 10.1145/366413.364612 (cited on page 11).

[169] Run and cycling tracking on the social network for athletes. URL: https://www.strava.com/ (cited on page 106).

[170] A. Sasturkar, R. Agarwal, L. Wang, and S. D. Stoller. Automated Type-Based Analysis of Data Races and Atomicity. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '05, pages 83–94, New York, NY, USA. Association for Computing Machinery, 2005. ISBN: 1595930809. DOI: 10.1145/1065944.1065956 (cited on page 1).

[171] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau. Noise Injection Techniques to Expose Subtle and Unintended Message Races. PPoPP2017, Feb. 2017. URL: https://kento.github.io/files/2017-02-06-PPoPP2017-conf-slides.pdf (cited on page 42).

[172] K. Sato, D. H. Ahn, I. Laguna, G. L. Lee, M. Schulz, and C. M. Chambreau. Noise Injection Techniques to Expose Subtle and Unintended Message Races. *SIGPLAN Not.*, 52(8):89–101, Jan. 2017. ISSN: 0362-1340. DOI: 10.1145/3155284.3018767 (cited on pages 6, 7, 42, 110).

[173] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a dynamic data race detector for multithreaded programs. *ACM Trans. Comput. Syst.*, 15(4):391–411, Nov. 1997. ISSN: 0734-2071. DOI: 10.1145/265924.265927 (cited on pages 1, 12, 19–22).

[174] M. L. Scott. Shared-Memory Synchronization. *Synthesis Lectures on Computer Architecture*, 8(2):1–221, June 2013. DOI: 10.2200/s00499ed1v01y201304cac023 (cited on pages 8, 11, 17).

[175] Seer-Lab. Ccmetrics: a static analysis tool for calculating concurrency code metrics in java programs. URL: https://github.com/seer-lab/ccmetrics (cited on pages 80, 95).

[176] K. Serebryany and T. Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, pages 62–71, New York, NY, USA. Association for Computing Machinery, 2009. ISBN: 9781605587936. DOI: 10.1145/1791194.1791203 (cited on page 21).

[177] A. Shoshani and E. G. Coffman. Sequencing Tasks in Multiprocess Systems to Avoid Deadlocks. In *SWAT*, 1970 (cited on page 15).

[178] M. Shousha, L. Briand, and Y. Labiche. A UML/MARTE Model Analysis Method for Uncovering Scenarios Leading to Starvation and Deadlocks in Concurrent Systems. *IEEE Transactions on Software Engineering*, 38(2):354–374, 2012. DOI: 10.1109/TSE.2010.107 (cited on page 24).

[179] Soot OSS. Soot-oss/soot wiki - github. 2022. URL: https://github.com/soot-oss/soot/wiki (visited on 02/14/2022) (cited on page 63).

[180] A. Srivastava, A. Edwards, and H. Vo. Vulcan: binary transformation in a distributed environment. (MSR-TR-2001-50):12, Apr. 2001. URL: https://www.microsoft.com/en-us/research/publication/vulcan-binary-transformation-in-a-distributed-environment/ (cited on page 36).

[181] J. Stasko. Animating algorithms with xtango. *SIGACT News*, 23(2):67–71, May 1992. ISSN: 0163-5700. DOI: 10.1145/130956.130959 (cited on page 138).

[182] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Protocol for Internet Applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, Feb. 2003. ISSN: 1063-6692. DOI: 10.1109/TNET.2002.808407 (cited on page 7).

[183] T. Su, K. Wu, W. Miao, G. Pu, J. He, Y. Chen, and Z. Su. A Survey on Data-Flow Testing. *ACM Comput. Surv.*, 50(1), Mar. 2017. ISSN: 0360-0300. DOI: 10.1145/3020266 (cited on page 22).

[184] K. Tai. Testing of concurrent software. In *[1989] Proceedings of the Thirteenth Annual International Computer Software & Applications Conference*. IEEE Comput. Soc. Press, 1989. DOI: 10.1109/cmpsac.1989.65057 (cited on page 17).

[185] K.-C. Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *1994 Internatonal Conference on Parallel Processing Vol. 2*, volume 2, pages 69–72, 1994. DOI: 10.1109/ICPP.1994.84 (cited on pages 23–25).

[186] K.-C. Tai. Race analysis of traces of asynchronous message-passing programs. In *Proceedings of 17th International Conference on Distributed Computing Systems*. IEEE Comput. Soc. Press, 1997. DOI: 10.1109/icdcs.1997.598047 (cited on page 42).

[187] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *Proceedings of the 7th Workshop on Parallel and Distributed Systems Testing, Analysis, and Debugging - PADTAD '09*. ACM Press, 2009. DOI: 10.1145/1639622.1639629 (cited on page 31).

[188] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot - a java bytecode optimization framework. In *Proceedings of the 1999 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '99, page 13, Mississauga, Ontario, Canada. IBM Press, 1999 (cited on pages 35, 62, 64).

[189] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan. Soot. In *CASCON First Decade High Impact Papers on - CASCON '10*. ACM Press, 2010. DOI: 10.1145/1925805.1925818 (cited on page 35).

[190] M. van den Brand, A. van Deursen, J. Heering, H. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. Olivier, J. Scheerder, J. Vinju, E. Visser, and J. Visser. The asf+sdf meta-environment. *Electronic Notes in Theoretical Computer Science*, 44(2):3–8, June 2001. DOI: 10.1016/s1571-0661(04)80917-4 (cited on page 35).

[191] W. Visser, C. S. Păsăreanu, and S. Khurshid. Test input generation with java pathfinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004. ISSN: 0163-5948. DOI: 10.1145/1013886.1007526 (cited on pages 26, 87).

[192] C. Von Praun and T. R. Gross. Static Detection of Atomicity Violations in Object-Oriented Programs. *J. Object Technol.*, 3(6):103–122, 2004 (cited on pages 20, 21).

[193] D. Padua, editor. *Race Detection Techniques*. Boston, MA, 2011. Springer US, pages 1697–1706. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_38 (cited on page 20).

[194] C. Wang, R. Limaye, M. Ganai, and A. Gupta. Trace-Based Symbolic Analysis for Atomicity Violations. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 328–342. Springer Berlin Heidelberg, 2010. DOI: 10.1007/978-3-642-12002-2_27 (cited on page 21).

[195] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *Proceedings 20th IEEE International Conference on Distributed Computing Systems*, pages 464–474. IEEE, 2000 (cited on page 7).

[196] A. Williams, W. Thies, and M. D. Ernst. Static Deadlock Detection for Java Libraries. In *ECOOP 2005 - Object-Oriented Programming*, pages 602–629. Springer Berlin Heidelberg, 2005. DOI: 10.1007/11531142_26 (cited on page 23).

[197] W. Winkler. String comparator metrics and enhanced decision rules in the fellegi-sunter model of record linkage. *Proceedings of the Section on Survey Research Methods*, Jan. 1990 (cited on pages 59, 74, 106).

[198] R. Xue, X. Liu, M. Wu, Z. Guo, W. Chen, W. Zheng, Z. Zhang, and G. Voelker. Mpiwiz: subgroup reproducible replay of mpi applications. *SIGPLAN Not.*, 44(4):251–260, Feb. 2009. ISSN: 0362-1340. DOI: 10.1145/1594835.1504213. URL: https://doi.org/10.1145/1594835.1504213 (cited on page 110).

[199] D. Yan, G. Xu, and A. Rountev. Rethinking soot for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis - SOAP '12*. ACM Press, 2012. DOI: 10.1145/2259051.2259053 (cited on page 64).

[200] M. Young and R. Taylor. Rethinking the Taxonomy of Fault Detection Techniques. In *11th International Conference on Software Engineering*, pages 53–62, 1989. DOI: 10.1109/ICSE.1989.714393 (cited on page 19).

[201] M. Yu, J.-S. Lee, and D.-H. Bae. AdaptiveLock: efficient hybrid data race detection based on real-world locking patterns. *International Journal of Parallel Programming*, 47(5-6):805–837, June 2018. DOI: 10.1007/s10766-018-0579-5 (cited on page 41).

[202] M. Yu, Y.-S. Ma, and D.-H. Bae. Efficient noise injection for exposing hidden data races. *The Journal of Supercomputing*, 76(1):292–323, Oct. 2019. DOI: 10.1007/s11227-019-03031-0 (cited on pages 1, 28, 40).

[203] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar. A platform for secure static binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments - VEE '14*. ACM Press, 2014. DOI: 10.1145/2576195.2576208 (cited on page 33).

[204] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting Severe Concurrency Bugs through an Effect-Oriented Approach. *SIGPLAN Not.*, 45(3):179–192, Mar. 2010. ISSN: 0362-1340. DOI: 10.1145/1735971.1736041 (cited on page 21).

[205] S. Zhou, M. Stumm, K. Li, and D. Wortman. Heterogeneous distributed shared memory. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):540–554, 1992. DOI: 10.1109/71.159038 (cited on page 6).

Appendix A

IBM Programs Descriptions

A.1 account

The `account` program simulates a banking system, with multiple accounts simultaneously making deposits, withdrawals and transfers. Each of these accounts is managed by a thread, with all threads executing these operations concurrently.

In this program, synchronization is achieved via object locks implemented via Java's synchronized region concurrency primitive. However, when executing a transfer between two accounts, only one lock is obtained, which may result in a high-level data race. Additionally, there is a non-atomic incrementation of an account's balance.

A.2 airlinestickets

The `airlinestickets` program simulates an airline ticket selling system, with multiple threads attempting to acquire their own ticket from a finite supply of tickets.

Threads for sales are invocated in a loop, and each time a sale is made, the number of sales is updated and checked non-atomically. This means that, in the end, there may be a higher number of sales than there were tickets available.

A.3 allocationvector

The `allocationvector` program manages allocations of values to a byte array. Two threads have a set number of values to allocate and do so concurrently. However, the programmer purposefully added a considerable delay, via a `for` loop, between the threads' spawns. Which, makes any concurrency bug inherently latent.

The bug occurs because, when allocating a value, it first finds a free block and only then allocates it, with these two steps being executed non-atomically.

A.4 boundedbuffer

The `boundedbuffer` program manages a buffer and producers and consumers which access it, to insert and remove values. All of these producers and consumer are implemented as threads which are listening on the buffer's monitor. The buffer periodically emits a `notify` signal, which randomly wakes up either a producer or consumer, depending on the state of the buffer. The producer or consumer thread completes its routine and then suspends.

A bug occurs because, since notify is used, a producer may be awakened when the buffer is full or a consumer when the buffer is empty. When this situation occurs, the instance cannot access the buffer, which in turn causes a `notify` signal to trigger, a deadlock occurs.

A.5 bubblesort

The `bubblesort` program is an implementation of the classic BubbleSort algorithm which creates a thread for executing each step of the sorting.

The bug occurs when a thread, executing a new step of the algorithm, starts before the last thread, representing the previous step, has finished.

A.6 bubblesort2

Another implementation of the BubbleSort algorithm, the `bubblesort2 program` is similar to the previous. However, it guarantees the ordering of the BubbleSort threads via a sleep statement which occurs when each thread is initialized.

The bug should not occur in a non-busy and non-adulterated system, as the developers mentioned and verified. But, when more noise is added to the program, different interleavings begin to occur.

A.7 bufwriter

The `bufwriter` program manages a buffer with producers and consumers, implemented as threads. Every time a producer or consumer writes or reads a value from the buffer, it increments the buffer's and the thread's operation counters.

The bug happens due to the fact that the programmer deliberately created an alternative producer whose operations are unsynchronized. This unsynchronized producer has a probability of being spawned instead of its synchronized variant, meaning the program may not manifest an error in many interleavings. Although it might not be a realistic programming pattern, this error is very hard to trigger, making a noise injection tool, such as OSCAR, all the more useful.

A.8 critical

The `critical` program has two threads accessing a critical section simultaneously. Before entering this section, they enable a flag to warn other threads that they will proceed.

The bug occurs because reading and writing operations to this flag are non-atomic, meaning it is possible to reach a deadlock, where both threads are forever waiting for it to be their turn. The original implementation of this program has no mechanism to check if it is currently deadlocked. As such, one such simple mechanism was added, creating another thread which waits a few seconds and terminates the program with an error, upon reaching this timeout.

A.9 dcl

The `dcl` program takes its name after the concurrent software anti-pattern it tries to reproduce (Double Checked Locking). The program is essentially s a system where travel agents, implemented as threads, reserve seats on a plane.

The concurrency error that this program contains is very subtle. Listing A.1 contains a snippet of the code in the precise location of the bug.

```
public void run() {
    for (int i = 0; i < seats_num; i++) {
        if (seats[i].ticket == null) {
            synchronized (seats[i]) {
                if (seats[i].ticket == null) {
                    seats[i].ticket = new Ticket(this.name);
                }
            }
        } else {
            synchronized (seats[i]) {
                check_ticket_details(i);
            }
        }
    }
}
```

Listing A.1: ”dcl” program concurrency bug.

The code in Listing A.1, refers to the loop that the travel agents execute concurrently to sell tickets. We see that both accesses to the ticket i are protected by a lock, which makes it seem correct. However, there is a moment, after assigning a new `Ticket` and before the `Ticket` object has been fully initialized, where the lock to i, has already been released. If another thread checks the ticket information before the constructor finished, the information obtained by it will become stale and, therefore, incorrect.

A.10 deadlock

The `deadlock` program creates a pool of threads that concurrently attempt to access, a pair at a time, a series of resources.

The bug occurs, because any access to two resources A and B, are not ordered. This means that two threads, one requesting $\{A, B\}$ and the other $\{B, A\}$, will eventually reach a deadlock.

A.11 deadlockexception

The `deadlockexception` program creates a pool of threads that do arbitrary division operations. A system is in place in order to avoid more than a set amount of threads to run simultaneously.

The division operation contains a small probability of triggering a division by zero error. This makes the thread stop unexpectedly, without decrementing the currently running thread counter, meaning less threads can run simultaneously. In a worst-case scenario, a deadlock may even be reached, when the number of failed threads is greater than or equal to the maximum number of simultaneously running threads.

A.12 garagemanager

The `garagemanager` program simulates a garage management system, dealing with worker tasks. The program first creates a pool of five workers, implemented as threads, starts their respective thread routine and the main thread waits for all of the workers' job completion.

However, there is a small probability that a worker is never given a job, and a manager is left waiting eternally, resulting in a deadlock.

A.13 linkedlist

The `linkedlist` program is an extended version of Java's built-in `LinkedList` Collection, offering synchronization when manipulating the list's contents.

The bug is present in the function which adds an element to the end of the list. Since this operations lacks synchronization, it becomes possible for a thread to retrieve a stale value right before and overwrite this last inserted element. In the end, it becomes trivial to verify if this happened, since it is just a matter of verifying the final list size.

A.14 liveness

The `liveness` program simulates an HTTP client-server interaction. The server, however, is set to only deal with a small amount of users simultaneously. If a client detects the server is too busy to accept its request, it will suspend and wait. Clients, however, are the ones responsible for checking, whenever their routine ends, if any suspended clients exist.

If a client is told by the server to suspend and is delayed enough time, between being rejected and added to the suspended queue, and every other thread finished, a starvation error will occur, with the thread left waiting.

A.15 lottery

The `lottery` program creates a pool of users, represented as threads, and, for each of these users, generates and assigns a random number.

A problem arises because the programmer uses a shared variable, as an intermediate variable to temporarily store the result of a pseudorandom number generator (PRNG). Since accesses to this variable are not synchronized, a user can end up with a number different from the one it was supposed to get and two users can even end up with the same number.

A.16 manager

The `manager` program simulates a memory management system, where a main thread (Memory Handler), is in charge of managing the release of memory blocks by other threads.

For a worker thread to signal the Memory Handler thread that its memory block release is complete, it uses a flag. The Memory Handler then acknowledges the flag and lowers it. While the flag is up, the worker is in a busy-waiting loop. However, accesses to the flag are not properly synchronized, meaning there is a chance that the flag will never go down, due to a data race. This results in deadlocks.

A.17 mergesort

The `mergesort` program is an implementation of the Merge Sort algorithm, which itself is an application of the divide-and-conquer parallelization strategy. The program contains a hardcoded integer array which is meant to be sorted by a series of spawned threads, whose number depends on the amount of set concurrency.

The program's algorithm contains a module which acts as a thread pool, keeping track of the amount of currently spawned threads which are currently either free or busy and assigning them tasks upon demand. The Merge Sort implementation checks the amount of available threads and then, if a thread is free, assigns it a task. Unfortunately, even though both of these steps are synchronized, the two-step operation itself is not, resulting in a high-level data race when two threads retrieve the same amount of available threads.

A.18 mergesortbug

The `mergesortbug` program is another implementation of the Merge Sort algorithm which has a very similar routine to `mergesort`. The program's bug is also very similar, resulting from repeated

unsynchronized accesses to the variable that contains the information on the current amount of available threads.

A.19 pingpong

The `pingpong` program spawns a series of threads after assigning a new reference (a ping-pong players) to a shared variable. These threads then access this variable concurrently and in multiple stages. First, they create a new reference and assign it the value of the shared variable. Then, they set the shared variable to null and, finally, after a delay, set the variable to the reference created in the first phase. The routine that executes all these phases is not synchronized.

A bug occurs when another thread starts manipulating the same variable concurrently, causing null pointer exceptions when accessing the variable, due to an order violation.

A.20 piper

The `piper` program simulates a flight management system. It is implemented following the producer-consumer paradigm, where a shared buffer, representing a flight, is used to board and offboard passengers. Whenever the flight is fully booked, other threads enter a busy-wait loop. The plane thread will keep checking if it is currently full, and will send a `notifyAll()` signal when it is finally free.

A bug occurs because the plane thread only waits for a single time, instead of in a loop, whenever it checks if the plane is full. This results in instances in which multiple threads will wake up and fill the plane simultaneously, overloading the buffer.

A.21 producerconsumer

The `producerconsumer` program is, as its name suggests, a toy program based on the producer-consumer concurrency pattern. The program contains an equal number of producers and consumers. The producers keep adding the current time to a queue, and the consumers keep removing these entries from this same queue. While many server-client-pairs can be spawned, they will always only communicate between the two of them, and never with clients/servers in other pairs.

During operation, the server and client threads are racing with each other. However, the synchronization is made via a counter which is incremented even when no operation is made, with this scenario occurring when the client finds an empty queue. This essentially means that, if the client ever manages to get ahead of the server, the queue will eventually be found empty, resulting in an exception being thrown.

A.22 shop

The `shop` program simulates a scenario in which a store interacts with costumers and suppliers, via the producer-consumer concurrency pattern. Suppliers add goods to the shop, while costumers

remove (purchase) goods from the shop. The first step of the program routine, is to start a supplier thread to populate the store with goods. After this thread is started, the main thread invokes a `sleep()` and waits for 10 ms. After waking up, it spawns a series of costumers to consume the goods.

The bug occurs when the supplier does not terminate before the sleep time, causing costumers to be unable to purchase goods. Additionally, the costumers retrieve an item via a two-stage mechanism, first by atomically checking if the store is empty and, secondly, by atomically retrieving the object. Since these accesses to a shared variable are not inside an atomic region, a high-level data race may occur.

A.23 suns_account

The `suns_account` program simulates a banking system with many accounts executing a series of deposits on themselves. Each account executes a random number of deposits, saving the value both to its balance and to a shared integer which contains a sum of all the deposits made. In the end, the balance of all accounts is compared to the total sum of all deposits made, triggering an error when the values do not match.

Since all the accounts only update their own balance through deposits, there is no danger of a data race occurring in these operations. However, the update of the total sum of deposits, is not done atomically. This results in a latent bug, where the incrementation operation may be done over a stale value, which was modified by another account thread earlier.

A.24 xtangoanimation

The `xtangoanimation` program is, by far, the largest of all the tested programs. Additionally to its size, `xtangoanimation` relies on XTANGO [181], which is a real-world animation framework, making this program one of the most interesting ones in the entire IBM benchmark.

The program uses multiple threads to render a simple animation. However, there is a chance of a deadlock occurring during the rendering.

Appendix B

OSCAR Error Detection Results

B.1 The account Program

The results for the `account` program are shown in Figure B.1. OSCAR noise injection was successful in triggering erroneous interleavings, exposing a latent bug.

In this program, the `yield` noise primitive did not prove effective, with different intensities not producing witnessable differences in effectiveness.

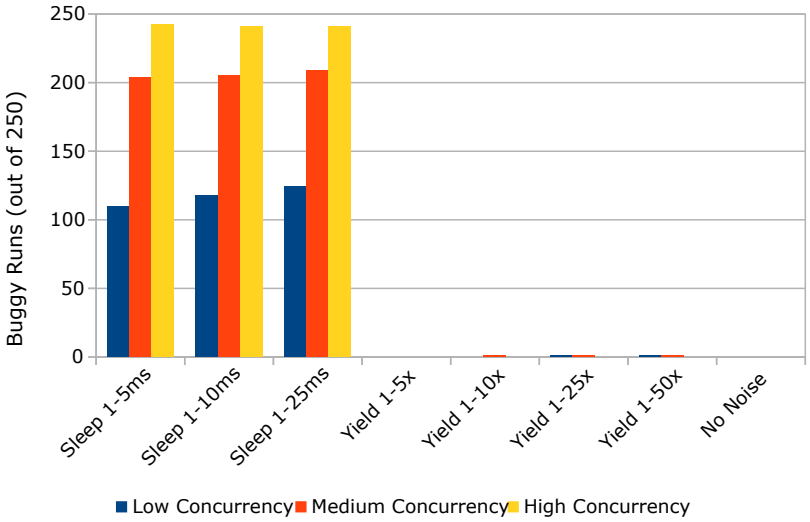


Figure B.1: account program error detection results.

B.2 The airlinestickets Program

The results for the `airlinestickets` program are shown in Figure B.2. OSCAR noise injection was not successful in raising the probability of triggering erroneous interleavings.

The `sleep` noise primitive proved much less effective than `yield` in this example. The reasoning for this result is due to the fact that the error stems from an unprotected access to a shared variable, in the condition of an `if` statement. Currently, OSCAR does not noise or track shared

variable dependencies stemming from conditional statements, meaning the noise placement necessary for triggering this error is not currently supported. This is the reason behind the `sleep` primitive performing so poorly, since it will just distance the thread accesses to this region more.

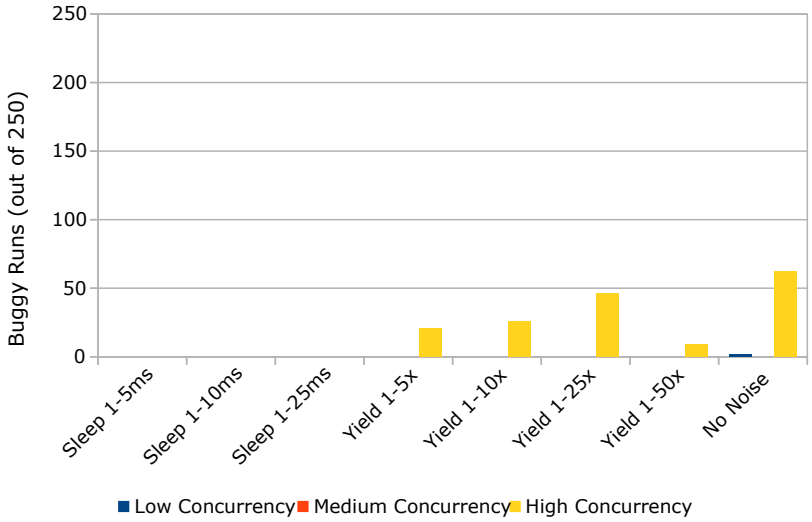


Figure B.2: airlinestickets program error detection results.

B.3 The allocationvector Program

The `allocationvector` program exhibited errors even without any noise being inserted, as its bug was not latent. Since the program runs through a loop with thousands of iterations, in which accesses to shared variables are made, the shared variable noising heuristics were used, as the tests would take an unacceptable amount of time to complete.

In Figure B.3, we can observe that injecting noise with OSCAR managed to increase the probability of encountering and erroneous interleaving.

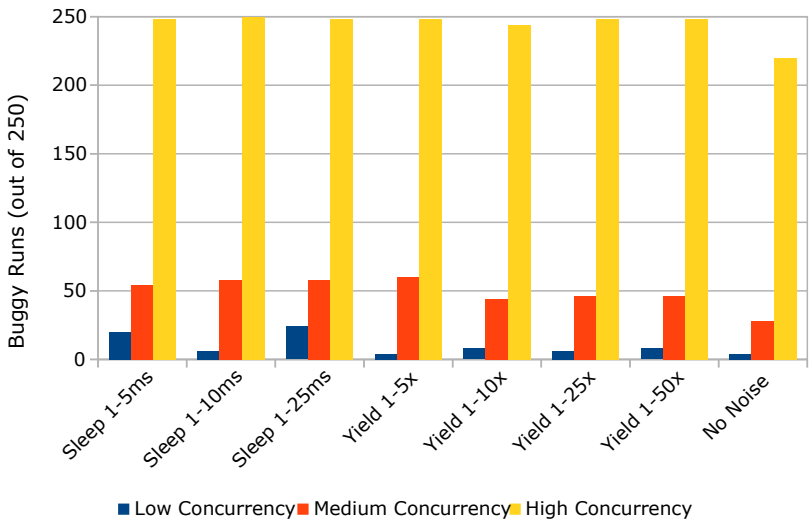


Figure B.3: allocationvector program error detection results.

B.4 The boundedbuffer Program

The concurrency error contained in the `boundedbuffer` program, was notoriously harder to trigger than the others previously tested, as can be evidenced by Figure B.4's y-axis' scale. With a low amount of contention, no erroneous interleavings were triggered, just as reported by the developers of the program, during their testing. However, with higher contention, OSCAR's noise injection managed to trigger erroneous interleavings with a probability of around 2%, exposing an otherwise latent bug.

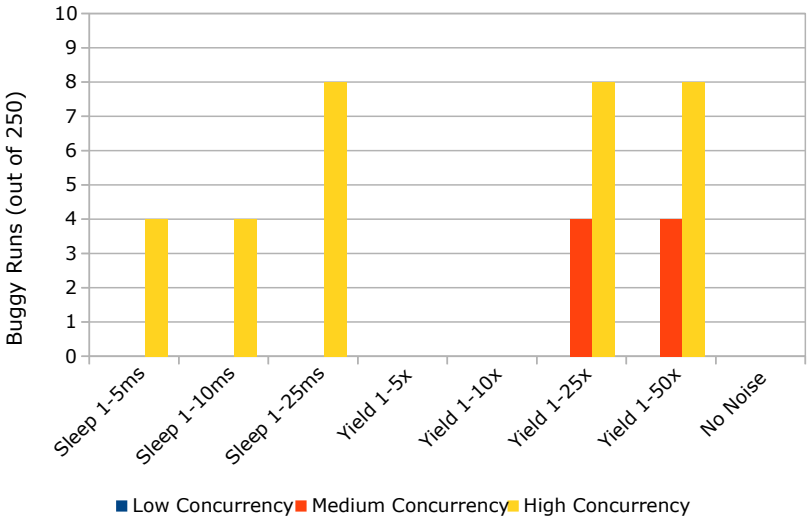


Figure B.4: boundedbuffer program error detection results.

B.5 The bubblesort Program

As evidenced by Figure B.5, OSCAR managed to expose an otherwise latent bug in the `bubblesort` program.

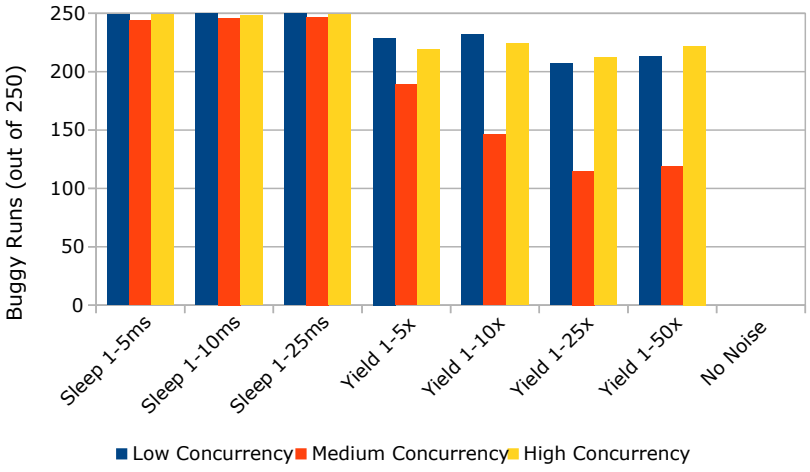


Figure B.5: bubblesort program error detection results.

B.6 The bubblesort2 Program

The `bubblesort2` program's implementation differed considerably from `bubblesort`'s. One key difference was the amount of shared variable accesses in a loop, making shared variable noising infeasible. Still, even though noise locations pertaining to this heuristic were disabled, OSCAR managed to highly raise the probability of triggering erroneous interleavings, even making it possible to evidence them in lower scenarios of lower than "high" contention. These results are shown in Figure B.6.

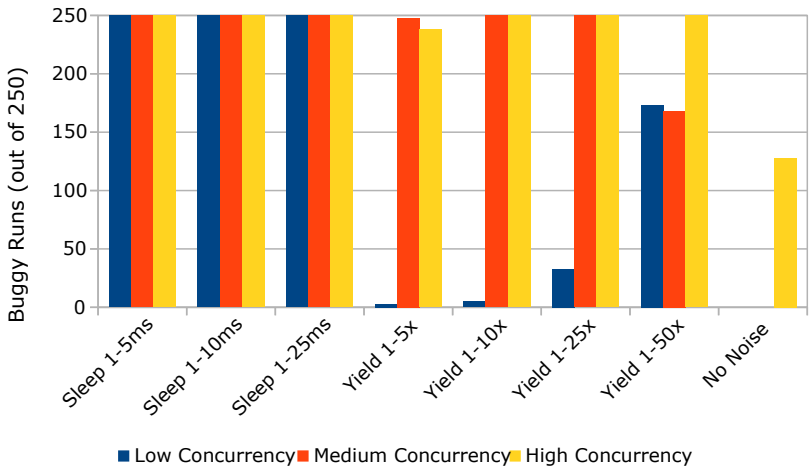


Figure B.6: bubblesort2 program error detection results.

B.7 The bufwriter Program

The `bufwriter` program has an unrealistic method of creating a latent bug in an otherwise correct program, as the chance for the bug to occur is tied to probabilistic function. Since the error occurred only under synchronization locations, only synchronized-based heuristics were enabled. Other heuristics ended up distancing the threads more, lowering contention in the synchronization locations. This phenomenon was also witnessable for higher noise intensities, which were detrimental for triggering erroneous interleavings, as illustrated in Figure B.7.

Overall, OSCAR managed to slightly increase the probability of triggering an erroneous interleaving, by around 15%, across all three levels of contention.

B.8 The critical Program

For the `critical` program, it was necessary to disable the injection of noise before the start of thread routines, as it distanced the two thread's routines and made them never enter the synchronization location, responsible for triggering the concurrency error, simultaneously.

After adding the deadlock detection avoidance mechanism to the original program, it was possible to run into a deadlock only once in all 250 runs, through each of the noise seeding

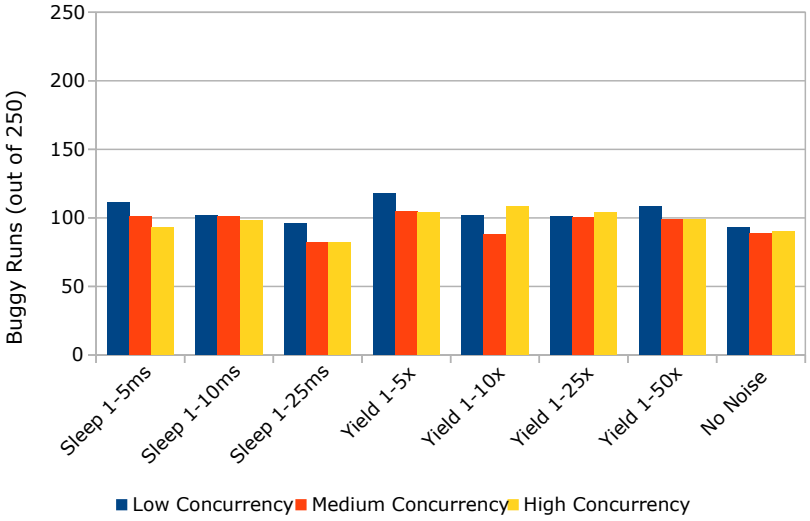


Figure B.7: bufwriter program error detection results.

settings. OSCAR managed to trigger this one erroneous interleaving using the `sleep` primitive with the noise intensity set between 1 and 25 ms.

B.9 The dcl Program

The `dcl` program contains an error-prone software pattern (Double Checked Locking), in which an access may be made to an object that, although is not null, may not be completely initialized.

None of the proposed heuristics for OSCAR supported this kind of error pattern. As such, a new simple heuristic was created, inspired by this specific error, contained in the `dcl` program. The heuristic inserts noise before the start of an object initialization, delaying the operations. The objective, is to give another thread enough time to do the unprotected access the uninitialized class, unearthing the latent bug.

Still, even this new heuristic could not trigger the erroneous interleaving, after trying thousands of times with different noise seeding settings, such as 1 to 25 ms. Likewise, the original developer also could not trigger any erroneous interleavings, attributing this to the bug being very hard to trigger and requiring specific JVM implementations. Given that the noise is being injected before the object's constructor first instructions, it should work. However, our analysis coincides with the original developer's — either context switches are not allowed inside constructors, or there is an implied locking mechanism until a constructor has completed.

B.10 The deadlock Program

Figure B.8 contains the error detection results for the `deadlock` program. OSCAR's noise injection managed to trigger the latent error by disturbing the regular scheduling.

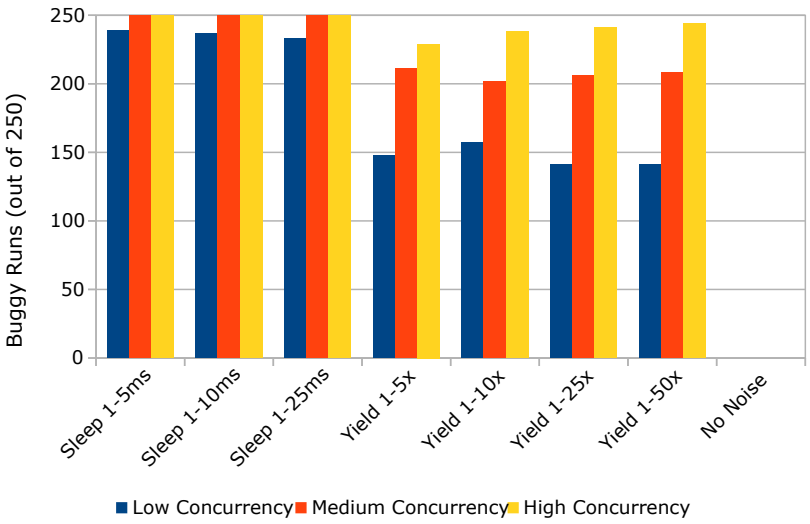


Figure B.8: deadlock program error detection results.

B.11 The deadlock_exception Program

The `deadlock_exception` program's error was dependent on a probabilistic event — a division by zero. This meant that any noising would be futile, as the error was not dependant on scheduling. As such, OSCAR's noise injection provided no advantage in triggering erroneous interleavings, as observable in Figure B.9.

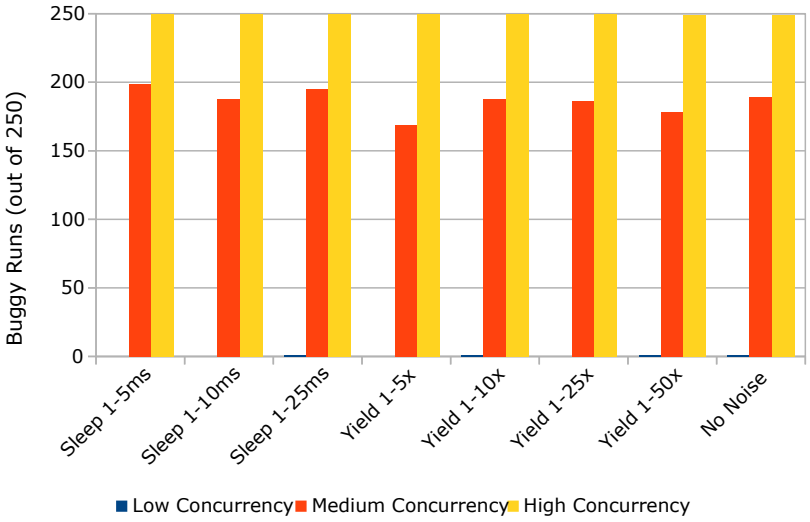


Figure B.9: deadlock_exception program error detection results.

B.12 The garagemanager Program

In order to test the `garagemanager` program, a deadlock avoidance mechanism had to be added to the original program. From Figure B.10, it is possible to visualize how OSCAR managed to severely raise the probability of triggering an erroneous interleaving.

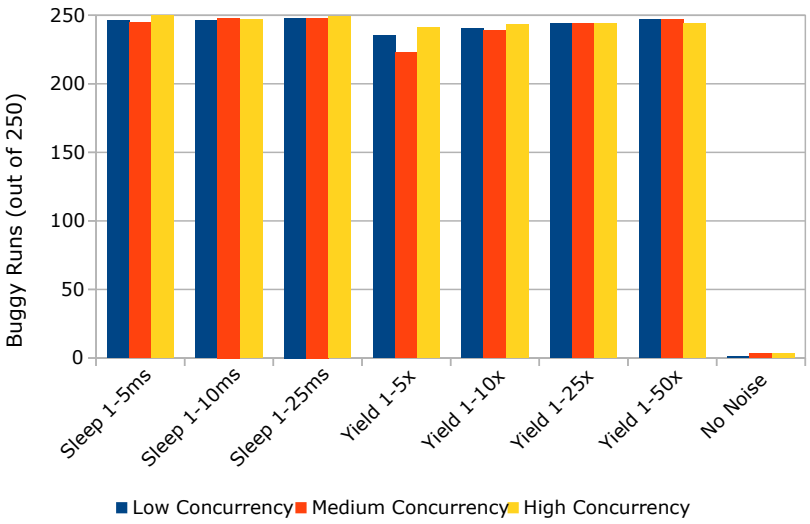


Figure B.10: garagemanager program error detection results.

B.13 The linkedlist Program

The `linkedlist` program, similarly to the `airlinestickets` program, whose results were discussed earlier in Section B.2, had a concurrency error stemming from an unprotected access to a shared variable, contained in an `if` statement. However, unlike that other program, `linkedlist`'s dependency stemmed from evaluating the value of a shared variable inside the condition of the `if` statement. This program shows that OSCAR should be extended to support this kind of dependency in its shared variable noising heuristic, at a later stage.

Since the program's routine only included thread cooperation at certain points of their routines (when both threads accessed shared buffer positions), it made sense to only add noise before the thread's routine, as to allow them to "meet" at the right moment and cause the erroneous interleaving.

Although OSCAR lacked complete support for the kind of heuristic present in the program, the above technique still managed to trigger erroneous an erroneous interleaving with less than a 0.5% chance, with both `sleep` and `yield` noise types.

B.14 The liveness Program

The `liveness` program contained a latent bug whose respective erroneous interleavings were made possible to trigger via the use of OSCAR. The overall results can be consulted in Figure B.11. The `sleep` noise type performed better, however only at a lower intensity. This can almost surely be attributed to the distancing caused by the use of a highly intense noise.

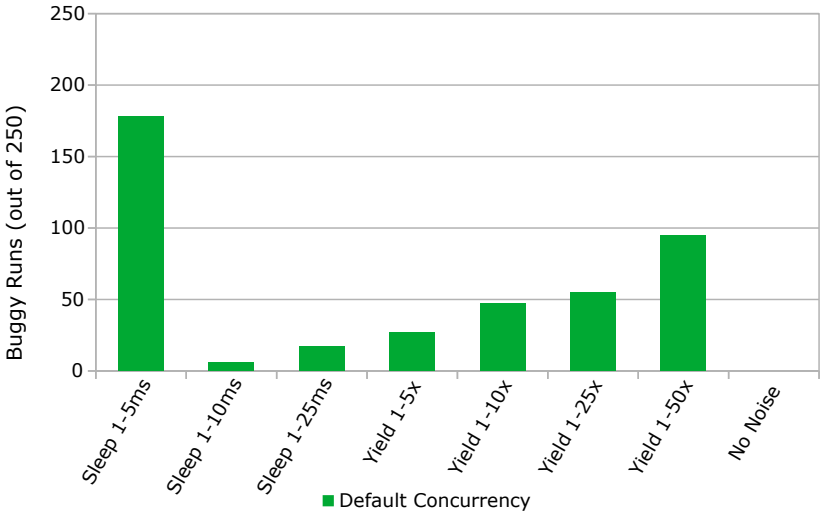


Figure B.11: liveness program error detection results.

B.15 The lottery Program

The error contained in the `lottery` program was always triggered, even without noise, as evidenced by Figure B.12. Interestingly, the use of the `yield` noise type, at low intensities, OSCAR managed to occasionally mask the error, by triggering different interleavings.

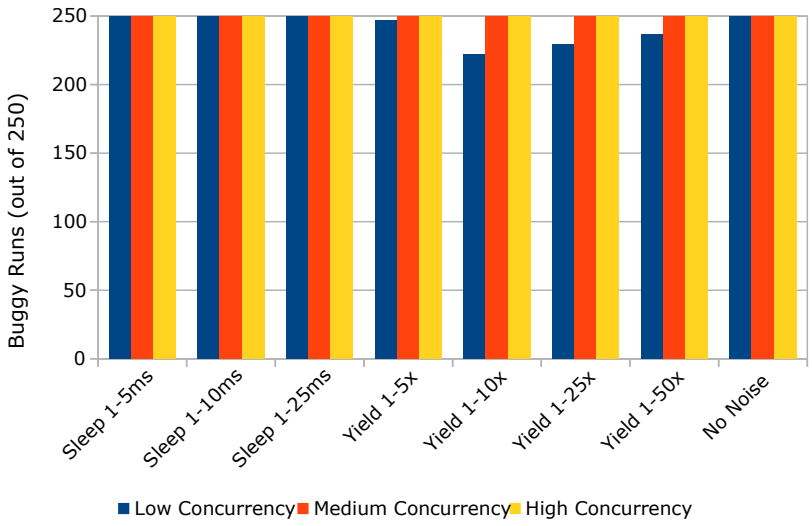


Figure B.12: lottery program error detection results.

B.16 The manager Program

Running an unmodified version of the `manager` program almost always resulted on a deadlock. Even though the program contained a deadlock error and a detection mechanism, these other deadlocks did not seem to be expected from the user, as they were not documented. To be able

to test the program further, it was necessary to adapt with an additional deadlock avoidance mechanism.

While the originally reported deadlock stemmed from improper accesses to a shared variable (a flag), the deadlock OSCAR managed to unearth, occurred due to an unexpected ordering in the thread initialization. The graph containing the total bugs detected, together with the amount of these bugs which were unexpected, extra, deadlocks, can be visualized in Figure B.13. Regular errors are, like the charts before, identified as "Low Concurrency" or "High Concurrency", while the extra deadlock errors found, which are a portion of the total errors, are identified as "Low Concurrency Deadlocks" or "High Concurrency Deadlocks"

Unlike most other programs, which possessed preset contention levels, the `manager` did not, allowing instead to control the number of spawned threads. As such, the amount of spawned threads chosen, after testing with various values, were of 5, 50 and 100 threads.

Overall, OSCAR was extremely successful in this test, as it managed to not only highly raise the probability of encountering erroneous interleavings, but also managed to find another previously undetected/undocumented error.

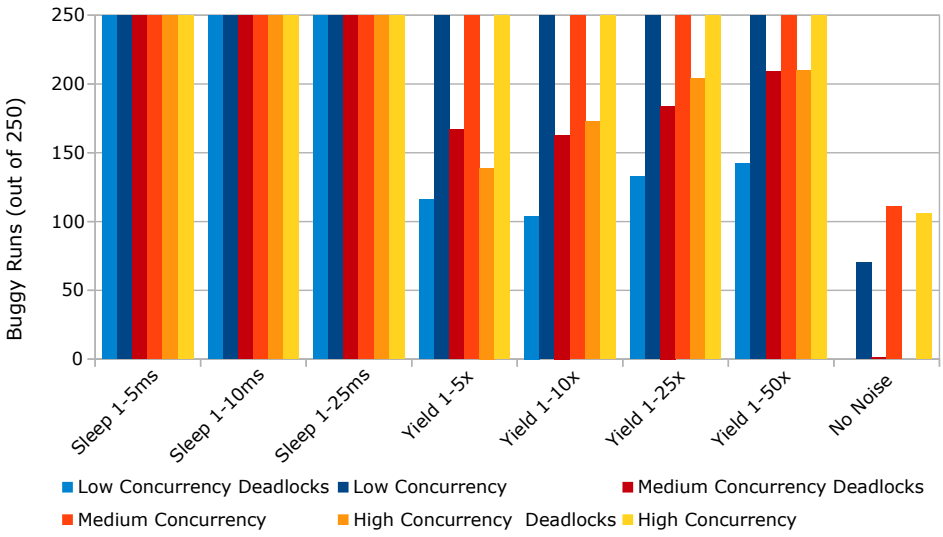


Figure B.13: manager program error detection results.

B.17 The mergesort Program

The different levels of contention chosen by the developer, for the `mergesort` program, exhibited extremely different results. At low contention, the unnoised program could not trigger any invalid interleavings, at medium contention, it managed to only once and, at high contention, it always triggered an invalid interleaving. This behaviour can be witnessed in Figure B.14. OSCAR managed to increase immensely the probability of triggering an erroneous interleaving at low and medium contention, with `sleep` performing a tad more efficiently at low contention.

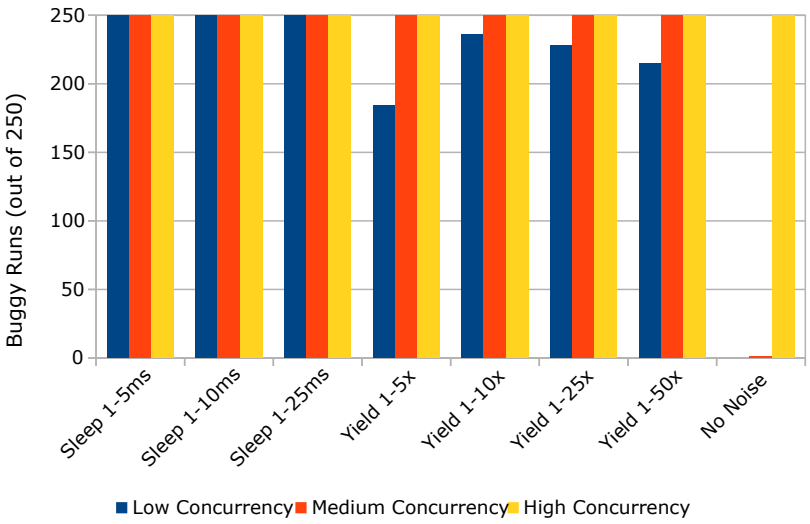


Figure B.14: mergesort program error detection results.

B.18 The mergesortbug Program

Originally, the `mergesortbug` program did not contain any preset concurrency levels. As such, the program was modified to accpet the configuration (array size and worker thread count) used while testing the `mergesortbug` program.

In Figure B.15, it is possible to visualize the results from testing the `mergesortbug` program. OSCAR's noise injection managed to trigger erroneous interleavings, exposing a latent bug. In lower contention settings, a lower intensity of `yield` noise showed the best results.

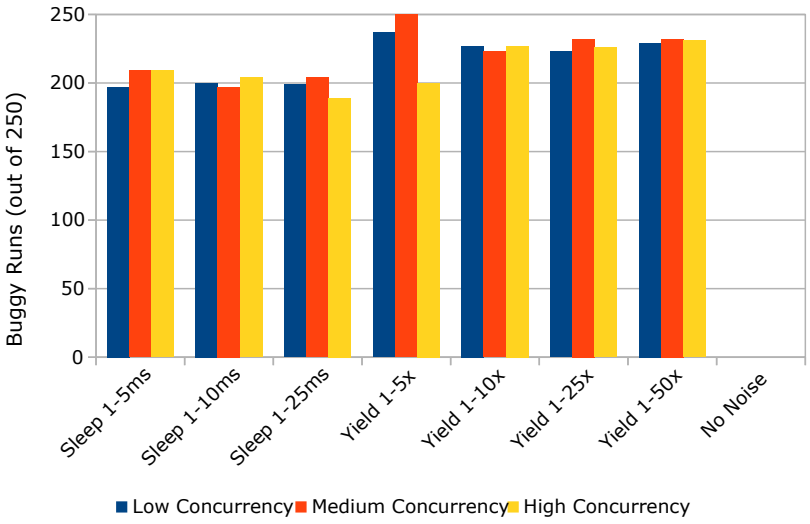


Figure B.15: mergesortbug program error detection results.

B.19 The pingpong Program

The `pingpong` program did not contain a latent bug, as unnoised runs would trigger the bug seemingly always. Figure B.16, presents the results obtained from testing the `pingpong` program.

While the `yield` noise type did not seem to have an effect on the results, the `sleep` noise type managed to trigger non-erroneous interleavings at lower contention levels. These results show that OSCAR noise injection managed to explore more of the program's state space.

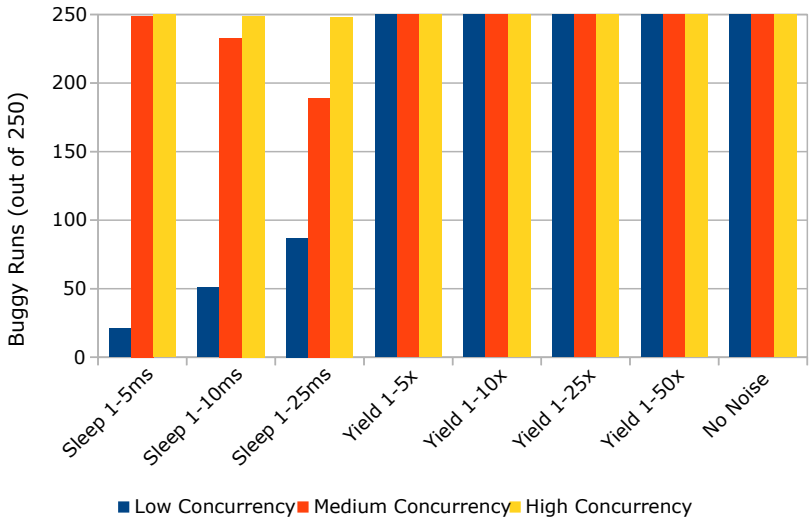


Figure B.16: pingpong program error detection results.

B.20 The piper Program

The results for the `piper` program tests can be consulted in Figure B.17. OSCAR successfully managed to trigger erroneous interleavings, exposing a latent bug. In this particular program, the `sleep` noise type was much more efficient at triggering erroneous interleavings.

B.21 The producerconsumer Program

The `producerconsumer` program's bug was not latent, triggering every single time, even when unnoised. However, with higher intensities of the `sleep` noise type, OSCAR managed to trigger non-erroneous interleavings. This shows that OSCAR's noise injection, through `sleep`, is managing to cover more of the program's state space.

B.22 The shop Program

The `shop` program had a latent bug which OSCAR managed to expose through noise injection, as can be visualized in Figure B.19, representing this program's test results. In this specific instance, the `yield` noise type was not able to trigger an erroneous interleaving.

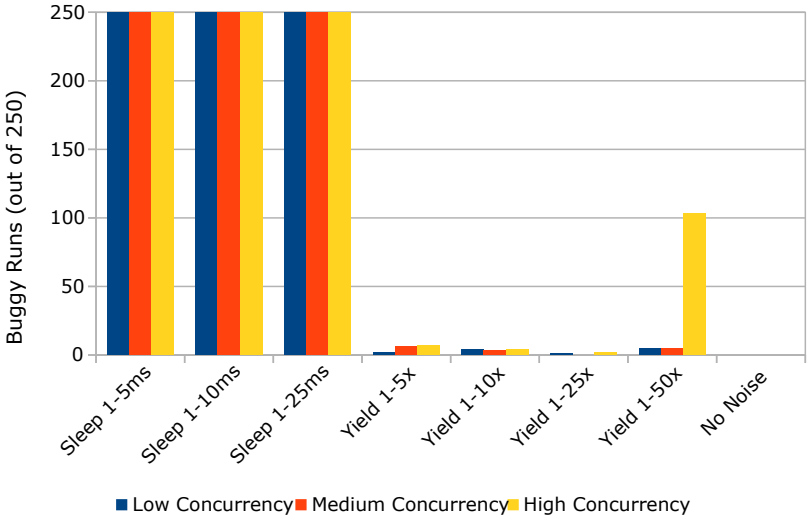


Figure B.17: piper program error detection results.

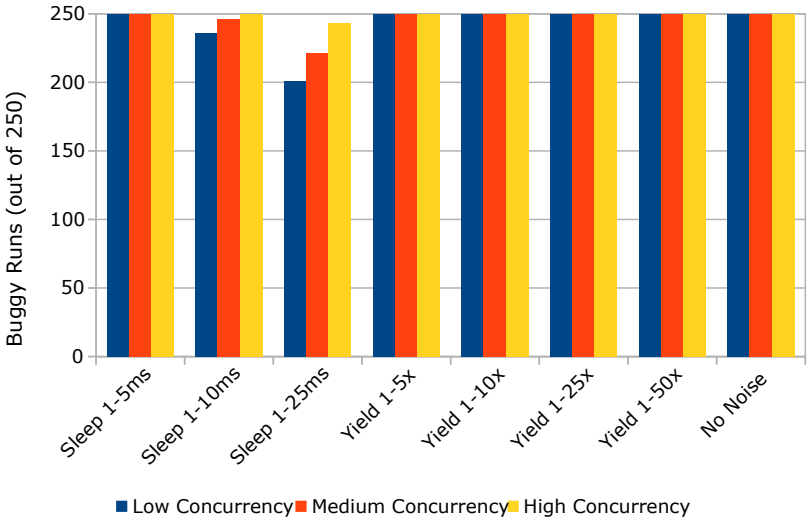


Figure B.18: producerconsumer program error detection results.

B.23 The suns_bank Program

The `suns_bank` program did not include preset configurations for concurrency levels, allowing instead a user to configure the number of account threads to be spawned. The amount of accounts chosen was 10, 100 and 1000, for low, medium and high concurrency levels, respectively.

This program's concurrency bug, although not latent, had quite a low probability of occurring. Overall, OSCAR managed to sharply raise the probability of triggering an erroneous interleaving. The respective results are illustrated in Figure B.20.

In this test's results, the `sleep` and `yield` noise types showed virtually equal levels of intensity.

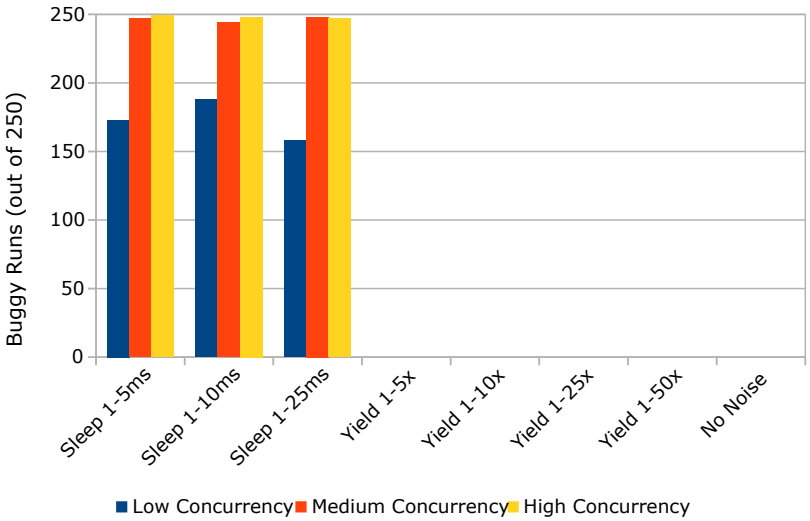


Figure B.19: shop program error detection results.

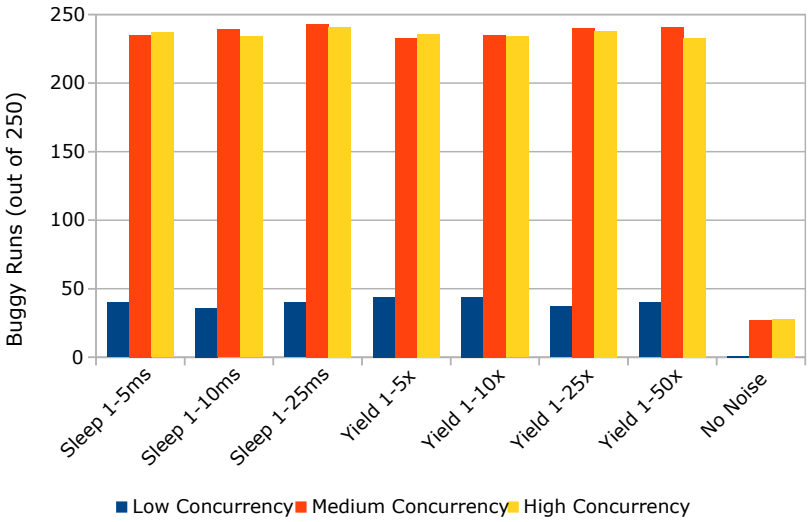


Figure B.20: suns_ bank program error detection results.

B.24 The xtangoanimation Program

The `xtangoanimation` program was the largest program, considering the amount of lines of code, in the entire IBM Concurrency Benchmark. When testing, it quickly became evident that OSCAR's noise injection was having an effect, as the program started deadlocking. Given how common deadlocks were, a deadlock avoidance mechanism had to be added to the program's code.

From Figure B.21, it is possible to ascertain that `xtangoanimation` had a latent bug, which OSCAR managed to expose through its noise injection engine. It seems apparent that the statements involving the bug are quite distant, given the much higher efficacy of the highest noise intensity of the `sleep` noise type, when compared to other noise types and intensities.

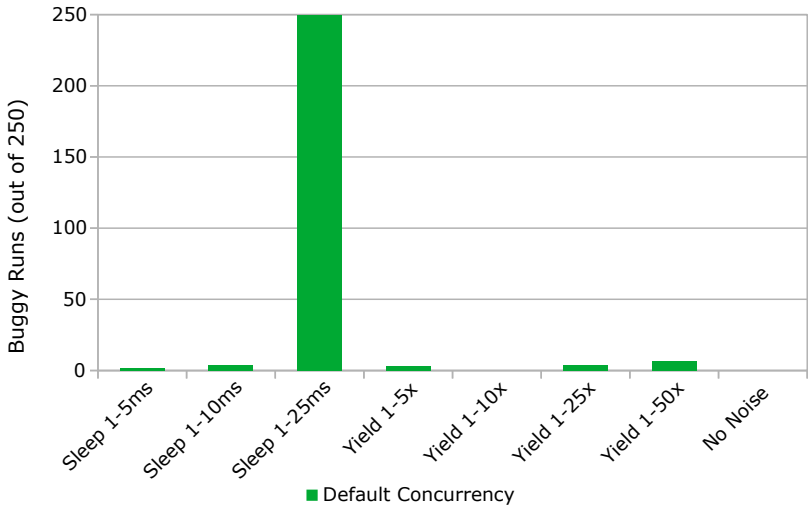


Figure B.21: xtangoanimation program error detection results.