VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALYSIS AND TESTING OF CONCURRENT PROGRAMS

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE ZDENĚK LETKO
AUTHOR

BRNO 2012

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INTELLIGENT SYSTEMS

ANALYSIS AND TESTING OF CONCURRENT PROGRAMS

ANALÝZA A TESTOVÁNÍ VÍCEVLÁKNOVÝCH PROGRAMŮ

DISERTAČNÍ PRÁCE
PHD THESIS

AUTOR PRÁCE Ing. ZDENĚK LETKO
AUTHOR

ŠKOLITEL prof. Ing. TOMÁŠ VOJNAR, PhD.
SUPERVISOR

ŠKOLITEL SPECIALISTA Ing. BOHUSLAV KŘENA, PhD.
CO-SUPERVISOR

BRNO 2012

Abstract

The thesis starts by providing a taxonomy of concurrency-related errors and an overview of their dynamic detection. Then, concurrency coverage metrics which measure how well the synchronisation and concurrency-related behaviour of tested programs has been examined are proposed together with a methodology for deriving such metrics. The proposed metrics are especially suitable for saturation-based and search-based testing. Next, a novel coverage-based noise injection techniques that maximise the number of interleavings witnessed during testing are proposed. A comparison of various existing noise injection heuristics and the newly proposed heuristics on a set of benchmarks is provided, showing that the proposed techniques win over the existing ones in some cases. Finally, a novel use of stochastic optimisation algorithms in the area of concurrency testing is proposed in the form of their application for finding suitable combinations of values of the many parameters of tests and the noise injection techniques. The approach has been implemented in a prototype way and tested on a set of benchmark programs, showing its potential to significantly improve the testing process.

Abstrakt

V disertační práci je nejprve uvedena taxonomie chyb v souběžném zpracování dat a přehled technik pro jejich dynamickou detekci. Následně jsou navrženy nové metriky pro měření synchronizace a souběžného chování programů společně s metodologií jejich odvozování. Tyto techniky se zejména uplatní v testování využívajícím techniky prohledávání prostoru a v saturačním testování. Práce dále představuje novou heuristiku vkládání šumu, jejímž cílem je maximalizace proložení instrukcí pozorovaných během testování. Tato heuristika je porovnána s již existujícími heuristikami na několika testech. Výsledky ukazují, že nová heuristika překonává ty existující v určitých případech. Nakonec práce představuje inovativní aplikaci stochastických optimalizačních algoritmů v procesu testování vícevláknových aplikací. Principem metody je hledání vhodných kombinací parametrů testů a metod vkládání šumu. Tato metoda byla prototypově implementována a otestována na množině testovacích příkladů. Výsledky ukazují, že metoda má potenciál vyznamně vylepšit testování vícevláknových programů.

Key Words

software testing, concurrency errors, dynamic analysis, verification, coverage metrics, noise injection, metaheuristics, genetic algorithms

Klíčová slova

testování software, chyby v souběžném zpracování, dynamická analýza, verifikace, metriky pokrytí, vkládání šumu, metaheuristické algoritmy, genetické algoritmy

Zdeněk Letko, Brno, 2012

Acknowledgement

First and foremost, I would like to thank my supervisor Tomáš Vojnar for his guidance and constructive criticism which have greatly helped me to progress towards the successful completion of this work. I would also like to gratefully thank Shmuel Ur for his motivating ideas and patience. My thanks also belong to Vendula Hrubá, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, and Bohuslav Křena who were always open to hear my ideas and to share their opinions. I must also thank to other PhD students in the A219 and A220 offices of FIT BUT who allow me to perform my time and resource demanding experiments on their personal computers. Finally, I would also like to express my deepest thanks to my parents for their support and to my wife Anna for her patience and tolerance without which I would never finish this dissertation.

The work presented in this thesis was supported by the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, the European FP6 project IST-035157 SHADOWS, and the internal BUT projects (FIT-10-1, FIT-11-1, and FIT-12-1).

Contents

Chapter 1

Introduction

The arrival of multi-core processors into regular computers accelerated development of software that uses multi-threaded design to utilize the available hardware resources. Modern operating systems allow hundreds of separate processes to run and communicate at any time. Switching among running processes requires a certain computation cost to be paid. Therefore, a lightweight version of processes were introduced. Threads, tasklets, and similar entities are lightweight processing units that can commonly be scheduled by the operating systems. Threads which exist within a process share process resources such as memory which makes communication among threads seemingly easier but, on the other hand, prone to errors. In this work, we primarily focus on problems that may arrise when multiple threads and/or processes use a common shared memory.

Concurrent or multi-threaded programming has become popular and supported by a wide range of programming languages including object-oriented languages like Java, C++, C#, and others. These languages allow programmers to relatively easily create multi-threaded programs. But, on the other hand, they put more demands on skills of programmers. Indeed, errors in concurrency in such languages are not only easy to cause, but also very difficult to discover and localize due to the non-deterministic nature of multi-threaded computation. Many different kinds of such errors exist. They can be broadly categorised according to the program properties they violate to *safety errors* such as data races, atomicity violations, deadlocks and *liveness errors* such as livelocks. A more detailed taxonomy of these errors is provided in Chapter 2.

The described situation stimulates research efforts which are currently devoted to all sorts of methods for discovering errors in concurrency (or, for proving their absence), including testing, dynamic analysis, as well as various approaches of formal verification. These approaches are briefly discussed in Section 1.1. Besides these approaches targeting quality assurance, the research also focuses on a new programming languages such as, e.g., Scala [102] or Go! [23], and new paradigms such as, e.g., transactional memory [97]. These languages and paradigms introduce concurrency mechanisms that allow to get the most out of modern multi-core processors while keeping the difficulty of programming on a reasonable level. Despite the advantages that these new approaches offer, the software industry accepts them very slowly mainly due to the fact that companies already invested a lot of effort, money, and human resources into products written in traditional programming languages. Therefore, testing, analysis, and verification of multi-threaded programs written in common languages like Java is still subject to heavy research. And, it is also the subject of this thesis.

The general subject of our interest in this thesis is finding concurrency errors in complex multi-threaded software systems written in object-oriented programming languages—especially in Java. In our approach, we focus on a combination of three already existing approaches described below: (i) noise injection that helps to increase the number of different scheduling scenarios that can be observed during repeated executions of a test case, (ii) dynamic analysis which is in certain cases able to detect errors based on information collected along an execution path even when the error does not show up, and (iii) stochastic optimisation algorithms and their application in the testing process.

Below, in the rest of this chapter, a brief overview of existing approaches for verification of multi-threaded software is provided. A special emphasis is devoted to program testing and dynamic analysis. Next, an overview of stochastic optimization algorithms is provided in Section 1.2. After this introduction, the goals of the thesis are described in Section 1.3. Finally, the structure of the following text is introduced in the last section of this chapter.

1.1 Verification of Multi-threaded Software

Plenty of research papers devoted to verification of multi-threaded software are published every year. The proposed approaches include program testing, dynamic analysis, static analysis, abstract interpretation, theorem proving, as well as model checking. We can compare these techniques by their ability to detect real errors. An ideal verification technique should be *sound* and *complete*. A sound technique does not produce any *false negatives* (i.e., it does not miss errors). A complete does not produce any *false positives* (i.e., warnings about errors that are not real and are sometimes called as *false alarms*). So, a sound and complete technique detects all and only real errors. Let us briefly introduce all mentioned techniques and focus on problems they have to face when multi-threaded programs are considered.

Program testing. Program testing is the most common way of finding errors in programs. The purpose of testing is to find program *failures*—situations when a program produces unintended results. Failures are caused by a software *defect* (also called as a fault or bug) which is a result of a programming error [24]. The process of finding and correcting of a program error that causes an observed failure is called *debugging*. When testing, a programmer or tester creates a *test case* which is defined by inputs and corresponding outputs. The test case is executed. If a failure occurs, there is an error in the program or in the test case. Program testing checks only the code along the execution path of the test case and usually does not provide information about the root cause of the failure which makes debugging difficult. In general, program testing is accepted as an unsound and complete technique.

Testing is often combined with *coverage analysis*—a process of collecting, reviewing and analysing *coverage metrics* [94]. A coverage metric is based on identifying phenomena (such as reachability of a certain line, reachability of a situation in which a certain variable has a certain value, etc.) whose occurrence in the behaviour of a tested program is considered to be of interest. More precisely, coverage metrics are formed using a *coverage domain* that is a set of *coverage tasks* representing the phenomena of interest. One can then measure how many of the phenomena corresponding to the coverage tasks have been seen in the execution(s) of the tested program. Such a measurement can be used to asses how well the program has been tested. Coverage metrics usable in concurrent software testing are discussed in more detail in Chapter 3.

The biggest problem of testing multi-threaded programs is that multiple executions of a multi-threaded program *MP* with the same input *I* may produce different results [93, 24] due to non-deterministic scheduling of program threads. Therefore, a single execution of *MP* is insufficient to determine the correctness of *MP* for *I*. Moreover, even if *MP* with *I* has been executed many times without spotting any failure, it is possible that a future execution of *MP* with *I* will produce an incorrect result. The non-determinism also complicates the debugging approach because there is no guarantee that if *MP* failed for *I* it will fail again during the next execution of *MP* with *I*.

This problem is targeted by various techniques of *program replay* that focus on a reproducible execution of multi-threaded programs, e.g., [56], and *deterministic testing*, e.g., [24], which attempts to force a selected scheduling scenario during a particular test execution. These techniques are in general quite demanding because they need to handle scheduling scenarios of possibly very long runs and to ensure that threads are scheduled according to particular scheduling scenarios. A lightweight alternative is to use *noise injection*, c.f., e.g., [30]. This technique randomly or based on some heuristics injects the so-called *noise* (a code that stops or delays the execution of a selected thread) in hope of observing a different and still legal scheduling scenario. Therefore, the approach cannot cause false positives apart from performance testing or testing of real-time programs for which this technique is not suitable. This approach cannot prove correctness of *MP* for the given *I*. But, it is efficient in revealing of program failures that are caused by real errors. Moreover, noise injection methods have a relatively low overhead in comparison with deterministic testing. Both mentioned approaches are described in more details in Section 2.3.

Dynamic analysis. Dynamic analysis which is also often called *runtime verification* [50] is based on *program tracing*—a process which gathers selected information concerning the observed execution. The gathered information is analysed with an intention to discover abnormal execution conditions. The information can be analysed *on-the-fly*, during the execution, or *post-mortem*, after the end of the execution. Despite the analysis gathers information concerning a single execution path, it can often discover errors that are not directly on the given execution path. An overview of existing dynamic analyses for concurrent programs is presented in Chapter 2.

Since dynamic analysis also performs execution of a test case, it suffers from the same problem of nondeterministic thread scheduling as testing does. An additional problem of multi-threaded program tracing is the so-called *probe effect* [24]—tracing of concurrent program interferes with the normal execution of the program. Therefore, designers and users of dynamic analyses must consider this effect during reasoning about the given program.

Static analysis. Static analysis [88] represents a different approach than the two introduced above. Static analysis is based on a *compile-time* analysis, and it often does not need the code to be executable. There exist many different static analyses ranging from rather simple which search the code for code patterns representing bad practices to quite complex complete and sometimes also sound analyses. Among the most well-known static analyses, the data-flow analyses, type and effect systems, and constrain-based analyses can be mentioned. Abstract interpretation and model checking introduced below are sometimes considered as static analyses too. We have, however, decided to list them separately because they represent rather specific techniques for program verification.

Static analyses, unlike testing and dynamic analyses, are not restricted to judging the behaviour of a program based on a few of its executions. They can, in theory, cover all possible behaviours. Moreover, that implies a need to somehow cover exponential many thread behaviours which makes precise analyses very costly. Therefore, static approach has to fight with exponential number of possible thread scheduling scenarios which makes analysis of multi-threaded programs quite hard. Therefore, there exist various static analyses which approximate thread behaviour. The more approximation they use, the bigger code they are able to analyse but for the price of producing more false alarms.

Abstract interpretation. Abstract interpretation introduced in [29] provides a theory of sound approximation of the semantics of computer programs. The semantics are based on monotonic functions over ordered sets, typically lattices. When using abstract interpretation, the code is symbolically executed under some abstract semantics, and for each line, abstract contexts forming an ordered set are computed. The semantics of a program is computed as the least fixpoint of the semantic function over the ordered sets. The abstract interpretation framework is generic and can be instantiated in a number of ways differing in their precision. The worst well-known instantiations are probably those for analysing integer programs using abstract domains such as polyhedra, octagons, difference bound matrices, etc. When used for verification of concurrent software, the abstract interpretation approach faces similar problems as static analysis.

Model checking. Model checking [12] is a fully automated technique for checking whether a system satisfies a certain correctness specification. The system can be a model prepared in some specialised modelling language (such as Promela), but also a real system (either hardware system in some hardware definition language or software system in some common programming language) possibly with abstracted environment. Based on a systematic or heuristic exploration of the state space of the examined system, the technique able to provide error traces explaining why a certain property does not hold in a given system. The technique is highly automated up to the possible creation of a model of the system and/or its environment.

The main problem of the technique is the so-called *state space explosion problem*, i.e., the number of states to which the system can get and that are to be explored is enormously huge. To cope with this problem, many different heuristics have been proposed. One of the sources of the problem is concurrency, and the number of possible scheduling scenarios leading to new states of the model. One of the heuristics limiting the number of scenarios to analyse is called *partial-order reduction* [43]. The heuristics exploits the commutativity of concurrently executed events which do not exhibit significantly different behaviours. Despite the existence of such heuristics, the state space of a concurrent program model can still be huge.

Theorem proving. Theorem proving which is sometimes called *deductive verification* [18] is an approach that is usually only semi-automated. The technique takes assumptions about the considered system and general theorems of various logical theories and using mathematical reasoning (and in some cases also hints from the user) proves that some facts are satisfied in the system. The process is usually complicated. Therefore, it is mostly used for critical systems where proof of correctness is really important. Recently, there has been a lot of progress in developing automated *satisfability solvers* for different logical theories. These solvers are used as a building block within various advanced verification methods. Together with heuristics for generation of loop invariants and function pre-/post- conditions, they can even allow fully-automated verification to be performed in some cases.

Theorem proving is not commonly used for verification of concurrent programs. This is mainly because applying this method to realistic sequential programs is already quite demanding. However, there exist recent pioneering works that apply theorem proving to concurrent programs, c.f., e.g., [2].

Despite the intense research in the area, deterministic testing, advanced static analyses, abstract interpretation, and model checking which are able to prove correctness of multi-threaded programs are still too demanding and do not scale well. Instead, simple static analyses, non-deterministic testing, and dynamic analysis are usually used by software developers and testers to search for errors in the code (this approach is sometimes called *bug hunting*). The bug hunting approach is often combined with the so-called *code inspection*. In this approach, skilled programmers read the code and manually search for common bad practices and possible errors. In this work, we primarily target and improve the bug hunting approach.

1.2 Metaheuristic Techniques

Stochastic optimisation techniques, also called *metaheuristics* or *search-based optimisation* [104, 76], employ a certain degree of randomness in the process of finding as optimal as possible solutions to complex well-defined problems. Such problems commonly have a large space of possible solutions (also known as *search space*) and no known efficient and complete solution. Instead, heuristics are used to partially explore the search space and favour promising parts of the space with good solutions. In order to be able to distinguish suitability of each solution, metaheuristic techniques define the so-called *fitness function* which is problem specific and express the quality of each candidate solution with respect to the chosen goal. With a metaheuristics approach, there is no guarantee to find globally optimal solutions. However, metaheuristics deliver satisfactory solutions for complex problems in a reasonable time.

The heuristic concept in solving optimisation problems was very popular in the past decades. Therefore, there exist many various techniques, c.f., e.g., the tabu search, simulated annealing, genetic algorithms, particle swarm optimization, etc. All of them combine two contradictory criteria: *exploration* of the search space and *exploitation* of the best solutions found.

The goal of exploration (also called *diversification*) is to try (very) different solutions so that all regions of the search space are evenly explored and that the search is not limited to a reduced number of regions. An example of an exploration algorithm is the *random search* algorithm [76]. In the random search, a random solution is generated in each iteration and no solutions are kept in memory for further use. If a good random generator is used, all regions of the search space are sooner or later explored.

The goal of exploitation of the best solution (also called *intensification*) is to search for a better solution in the region containing the currently most promising solution. An example of an exploitation algorithm is the *hill climbing search* algorithm [76]. In this algorithm, one selects the best neighbouring solution that is better than the current best solution in each iteration. This approach stores in memory the newly discovered solution as the current best solution for the next iteration.

Metaheuristic methods can be classified according to many different criteria, for instance, nature-inspired vs. non-nature-inspired, those which store solutions in memory for further use vs. those which do not (often denoted as memoryless), population-based vs. local, deterministic vs. stochastic, or iterative vs. greedy [104].

One of the popular techniques which is also used in our approach are the *genetic algorithms (GAs)* [76]. GAs are inspired by the evolution processes in nature, handle a set of solutions (called *population*) in memory, and during each iteration determine the next population using a stochastic algorithm containing three major steps called *selection*, *crossover*, and *mutation*. A solution in the genetic algorithm is encoded as a vector of values (usually but not necessarily of boolean values) called a *genome*. The selection operation decides which solutions (also called *parents*) will be used to infer a new member of the next population. The crossover operation combines two parents, and the mutation operation modifies (mutates) the result of the crossover. GA appropriately combines exploration and exploitation and often can find a good solution for the given problem.

It has been shown that metaheuristic techniques have many applications in most areas of software engineering such as test data generation (generation of test inputs), module clustering, cost/effort prediction, system integration, requirements scheduling, and non-functional properties testing such as security or usability [27, 81, 3]. Applications of metaheuristics in testing and verification of concurrent programs is discussed in Chapter 5.

1.3 Goals of the Thesis

As discussed above, at the present time, verification of multi-threaded code written in common programming languages like Java requires a considerable effort and skilled testers. The primary goal of the thesis is therefore to improve the testing process of real-world multi-threaded programs.

To achieve this goal, we propose new concurrency coverage metrics which are inspired by algorithms for dynamic detection of concurrency errors and appropriately handle important aspects of multi-threaded computation. They are suitable to measure quality of testing and to compare different testing approaches. Then, we focus on noise injection heuristics. We propose a new heuristics which uses coverage information that is collected during the testing process to further increase the number of examined scheduling scenarios. Moreover, we provide a systematic comparison of different noise injection heuristics based on a number of experiments and advise when and how to use them. Surprisingly, such information did not exist previously[1]. Finally, we propose a way how metaheuristic search algorithms can be used

[1]Personal and mail communication with Dr. Shmuel Ur (Shmuel Ur Inovations, Ltd., previously IBM Haifa Research Laboratories) and Dr. Yarden Nir-Buchbinder (Google, previously IBM Haifa Research Laboratories)—authors of the IBM Concurrency Testing Tool.

to improve the quality of testing of concurrency software by finding suitable combinations of the many parameters of test and noise techniques, which we formalise as the so-called test and noise configuration search problem. Our novel combination of the mentioned techniques leads to an efficient automatic technique for testing multi-threaded programs.

Besides the primary goal, this thesis also provides an original taxonomy of concurrency errors and an overview of techniques for their run-time detection which we studied before the main goal of the thesis has been tackled.

1.4 Structure of the Thesis

Chapter 2 provides an introduction to multi-threaded computation and then focuses on concurrency-related errors and dynamic methods of their detection. In Chapter 3, we present our new concurrency coverage metrics suitable mainly for search-based and saturation-based testing. Chapter 4 studies various existing noise injection techniques and introduces new noise injection techniques which are inspired by our work on concurrency coverage metrics. Chapter 5 presents our proposal of using metaheuristics in testing of concurrent programs. As with the experiments with noise heuristics, some more details of the experiments we did to set parameters of the genetic algorithm are presented in Appendix A. In Chapter 6, we conclude the work and provide directions of possible future work. Finally, in Appendix B, we describe an infrastructure for search-based testing of concurrent programs called SearchBestie that resulted from our work and that contains an implementation of the discussed techniques.

Chapter 2

Verification of Concurrent Programs

In this chapter, basic concepts and terminology used in the rest of this thesis are introduced. Namely, in the first section of this chapter, we introduce the basic principles of multi-threading and concurrent computation on modern systems with shared memory. Particular emphasis is put on the Java synchronisation mechanisms and on memory model describing the semantics of Java byte-code instructions accessing the memory. Programmers use these principles and mechanisms to design multi-threaded programs that keep the shared memory in a consistent state. To achieve this goal, their program has to follow certain rules in communication among threads. This high-level design of synchronisation is often called a *synchronisation policy* [93].

A violation of this policy leads to inconsistent content of the shared memory and to concurrency errors. Section 2.2 provides a taxonomy and uniform definitions of concurrency errors. Definitions of these errors surprisingly often differ in the present literature. Therefore, we try to identify the basic skeleton for each concurrency error and discuss its parameters. Our motivation is to describe similar terms present in the literature and connected with the same concurrency errors by appropriate setting of particular parameters. The errors are classified as safety and liveness errors according to the type of correctness criteria they violate.

Concurrency errors commonly manifest themselves only under certain interleaving of events in multiple threads. Therefore, it is often very difficult to spot them during an execution of a test. Section 2.3 presents two advanced approaches which can be used during the testing process to significantly increase the number of different examined interleavings without needing any further activity of the programmer. The noise injection technique presented in this section injects into the test execution instructions which influence decisions made by the scheduler. The second technique called deterministic testing controls decisions of the scheduler during the test execution and systematically explores the interleaving space of the program under test.

Still, even with the noise injection, it is often difficult to see the occurrence of an error. Therefore, the last Section 2.4 presents an overview of dynamic analysis techniques for detection of concurrency errors. These techniques can be combined with approaches described in Section 2.3 and they can further improve the testing process. The techniques are categorised according to concurrency errors they are able to detect.

2.1 Introduction to Multi-threaded Software

For each program that is executed, the *operating system* (OS) creates a *process* containing information about the execution such as process state (e.g., ready, running, waiting, or stopped), program counter, saved processor (CPU) register values, stack of activation records, memory management information, file descriptors, etc. The volume of this information can make it expensive to create and manage processes [24]. Therefore, in modern operating systems, each process contains one or more units of work called *threads* each handling a thread state (similar to process states), program counter, stack of activation records, and saved CPU registers. Memory management information, file descriptors, etc. are shared among all threads within a single process.

A sequential program is executed as a single process with a single thread of control. When it is executed with the same input, the sequential program goes through the same sequence of instructions and provides the same output[1]. This mostly deterministic behaviour makes analysis and verification of sequential programs simpler. On the other hand, a concurrent program is executed within one or more processes including multiple threads of control. For simplicity, only threads (potentially belonging to different processes) will be used there under.

In presence of multiple threads, the OS must decide how to allocate the CPU among them. The *scheduling policy* enforced by the OS subsystem called *scheduler* determines which of the threads in the ready state are selected for execution. Common operating systems use preemptive scheduling. In this approach, the selected thread is given a time slice of the CPU called a *quantum*. When the quantum expires, an interruption occurs. The scheduler is commonly called at the end of each interruption handler and can preempt the active thread to allow another thread to run. The thread is also allowed to *yield* the CPU before the quantum elapses. Switching the CPU from one thread to another is called a *context switch*. Modern schedulers are based on heuristic algorithms which together with a rather unpredictable occurrence of interruptions (due to the activity of users and the connected devices) make scheduling quite unpredictable and therefore *non-deterministic*.

[1]Assuming that the sequential program is not influenced by any source of non-determinism, e.g., it does not dependent on timing of some external events or run-time errors such as lack of memory, disk space, etc. Notice that pseudo-random generators provide deterministic output.

In an environment where multiple CPU cores exist, multiple threads can be executed at the same time. This is referred to as *true concurrency*. In most cases, however, the number of available cores is lower than the number of threads ready for execution. In such cases, the OS scheduler is switching ready threads and such execution is referred to as *pseudo-concurrent* execution or *interleaving concurrency* [77]. The interleaving concurrency is usually considered when reasoning about the behaviour of programs. This is based on an assumption that a true concurrent execution of two instructions is guaranteed by the underlying hardware to correspond to one of their interleaved executions.

Communication and Synchronisation. Non-deterministic scheduling explained above causes that the multi-threaded program executed with the same input multiple times may produce different outputs. For instance, imagine a program with two threads t_1 and t_2 producing the following output. Thread t_1 prints the string `hello` and thread t_2 prints the string `world` to the same output buffer. Assume that writing a character to the buffer is an *atomic operation* (it seems to the rest of the program to occur instantaneously), that threads have no exclusive access to the buffer, and the context switch can occur at any time. Then, the program can produce as its output all strings that preserve ordering of characters within these two words (due to the sequential nature of execution within each thread) such as `heworldlo`, `hweolrllod`, or `hellworldo`.

The example highlights how executed threads compete for a shared resource (the output buffer in this case). To fix the problem, each thread is to access the resource *exclusively* (or in *mutual exclusion*) [51]. When a thread is accessing a resource exclusively, the resource is *locked* to all other threads. Any thread that wants to access the resource must first lock the resource for other threads and unlock the resource when it is done with using the resource. If the resource is locked by some other thread, the thread has to wait until the resource is unlocked to proceed with locking the resource. This situation is called a *lock contention* and can negatively influence performance of multi-threaded programs. The code that is executed between the lock and corresponding unlock operations is often denoted as a *critical section*[2].

Utilisation of the described mutual exclusion technique can again be demonstrated on a simple example. If the code is modified so that each thread locks the output buffer before producing output and unlocks it when it is done, there is no possibility of interruption of the thread which is in the critical section by the thread which is waiting to obtain the lock. Therefore, only two possible outputs of the program are obtained: `helloworld` and `worldhello`.

The example shows a multi-threaded program where threads do not communicate with each other. Such programs are rare. Usually, threads work together and therefore need to communicate. In order to communicate, threads must *synchronise*

[2]The term critical section is more general [24].

their actions to share information. Processes and threads can communicate either by *message passing* through a common channel or by exchanging data using *shared memory* [13]. Message passing is often used for communication among multiple processes possibly located on different interconnected machines. The shared memory approach is preferred for communication among threads within a single process but can also be used for communication among processes that can access the same piece of memory. Shared memory is provided by the OS in this case.

The OS provides low-level synchronisation mechanisms such as *spin locks*, *mutexes*, *general semaphores*, *barriers*, etc. [13] which can be used to achieve mutual exclusion. Programming languages use these mechanisms to provide high-level synchronisation constructs, e.g., monitors and blocking queues [93]. These mechanisms can be used by programmers to easily achieve a specific synchronisation among threads.

This thesis primarily focuses on multi-threaded programs written in Java. Therefore, the rest of this section describes implementation of concurrency in Java, the Java memory model, and finally, basic synchronisation mechanisms available to Java programmers.

Multi-threading in Java

Java is a popular object-oriented programming language. Java code is compiled to intermediate code called *byte-code*. Bytecode is then executed within the Java run-time environment called Java Virtual Machine (JVM) which provides a uniform interface to the underlying platform [46]. Java inherently supports multi-threading. Each Java program has a main thread executing the `main()` method and represented by an object of type `Thread`. In addition, several system threads (e.g., a thread running the garbage collector) are started automatically whenever a Java program is executed. Therefore, every Java program is a concurrent program [24]. Java provides several methods to manage the lifetime cycle of threads—the most important are: `start()` (runs a thread), `join(t)` (waits for the thread *t* to end), `sleep(time)` (waits for a predefined *time*), `interrupt()` (interrupts thread), and `yield()` (yields the processor and lets the scheduler to act).

The Java memory model. The memory of JVM is divided into several run-time data areas. Some of them are created during JVM initialisation and are shared among all threads. Some are created for each thread and are exclusively accessed by these threads. A part of the shared memory that is important for verification of Java programs is called *heap*. The heap is shared among all threads and used by the JVM to store all class instances (objects) and arrays.

Each Java object can contain class and instance variables also called *fields*. The JVM operates on two kinds of variable types: *primitive types* (`byte`, `int`, `boolean`, etc.) and *reference types* [46]. Threads can therefore access either primitive or reference values stored in the heap. Access operations on these values are guaranteed to be atomic except for the primitive types `double` and `long` for which the JVM specification [46] allows each access to be treated as two atomic accesses.

The Java memory model guarantees the so-called *sequential consistency* for data race free programs (data races are described in the next section) [79]. The sequential consistency ensures that each read operation sees a value written by the last write operation that *happens before* it. The happens-before order is given by the transitive closure of the *program order* and the *synchronisation-with order*. The program order is defined by the ordering of instructions executed within a single thread. The Java compiler and JVM optimizers are allowed to reorder byte-code instructions as far as they preserve control and data dependencies as we know them from the theory of compilers, c.f., e.g. [5]. Therefore, the program order of the same code can slightly differ in different run-time environments and/or when different compilers are used.

The synchronisation-with order $\xrightarrow{sw}$ is a partial order over *synchronisation actions*. This partial order puts into relation two subsequent actions performed on the same synchronisation element. The source x of a synchronisation-with edge $x \xrightarrow{sw} y$ is called a *release*, and the destination is called an *acquire*. There are six scenarios defined in the specification [46] which induce the synchronised-with order:

i. An unlock action on a monitor m synchronises-with all subsequent lock actions on m.

ii. A write to a volatile variable v (volatile variables are introduced below) synchronises-with all subsequent reads of v by any thread.

iii. An action that starts a thread synchronises-with the first action in the thread that it starts.

iv. The write of the default value (`0`, `false`, or `null`) to each variable v synchronises-with the first action on v in every thread.

v. The final action in a thread t_1 synchronises-with any action in another thread t_2 that detects that t_1 has terminated (using the `isAlive()` or `join()` methods). (vi) If a thread t_1 interrupts thread t_2, the interrupt by t_1 synchronises-with any point where any other thread (including t_2) determines that t_2 has been interrupted (using the `interrupted()` or `isInterrupted()` methods).

The described memory model should be respected when designing analyses of multi-threaded Java program correctness. To get the whole picture, Java synchronisation mechanisms are introduced next.

Java synchronisation mechanisms. There are three basic synchronisation mechanisms in Java: Monitors, thread notification, and volatile variables. Some other more sophisticated synchronisation mechanisms such as barrier synchronisation or synchronised queues have been developed using these basic synchronisation mechanisms and are available in the `java.util.concurrent` package since Java 5 [93, 65]. In the following text, only the basic synchronisation mechanisms which are important for this thesis are described.

Each object in Java contains a monitor which allows one to implement locking and to achieve mutual exclusion. There are two ways how to obtain a lock. One way is to use the construct `synchronized(o){}` where a thread has to obtain the lock given by the object *o* before execution of the critical section in the braces. Another way is to declare a method as `synchronized`. Then the monitor of the instance or class containing such a method is used for synchronisation. Each thread has to acquire the monitor before the method execution. Monitor based mutual exclusion is sometimes called *implicit locking*. An implicit locking requires that the guarded block of code must be sequentially coded (a lock cannot be acquired in one method and released in another). This limitation can be avoided using the locks provided in the `java.util.concurrent` package. A use of such locks is sometimes called *explicit locking*.

Java monitors also allow threads to be synchronised by passing a notification message between them (sometimes called a *wait-signal* synchronisation). This can be accomplished by the *wait and notify* construction in Java. A thread which invokes method `wait()` on a selected monitor is suspended till some other thread invokes method `notify()` on the same monitor. An invocation of both of these methods must be guarded by the monitor that is used for waiting and notification. Therefore, the notification mechanism itself does not induce any synchronised-with relation. The relation is induced by locking operations on the monitor. The JVM specification allows a possibility of a spurious escape of a thread from the waiting state (due to an interruption or timeout). Because of this, a shared state variable should be used to control whether the thread was correctly notified.

Each field can be declared as *volatile*. For such fields, the JVM internally disables all caching functionality, and all accesses to volatile fields are atomic (even accesses to fields of type `long` and `double`). Changes are therefore done directly in the heap. This is often used for sharing a state among threads.

All concurrency mechanisms mentioned above can be used in a wrong way such that the compiler will not issue any warning. The program containing a concurrency error will hang or crash during the run time. The next section gives a general overview of what and how can go wrong if a wrongly implemented synchronisation is used.

2.2 Taxonomy of Concurrency Errors

This section strives to provide a uniform taxonomy of concurrency errors common in current programs that can help a better understanding of these errors and their possible treatment. Plenty of research papers describing new tools and techniques for detection of concurrency errors are presented each year. However, different authors describe the same concurrency errors in different terms and, surprisingly often, they even give the same concurrency error a different meaning. The inconsistencies in definitions of concurrency errors are often related to the fact that authors of various analyses adjust the definitions according to the method they propose. Sometimes the definitions differ fundamentally [37]. However, often, the definitions have some shared basic *skeleton* which is *parameterised* by different underlying notions (such as the notion of behavioural equivalence of threads). This is highlighted in the following taxonomy (originally published in [38]).

Of course, there have been several previous attempts to provide a taxonomy of concurrency errors in the past decades, cf., e.g., [71, 74, 21]. In [21], the authors focus on particular *error patterns* bound to particular synchronisation constructs in Java like, e.g., the `sleep()` routine call. In [74], a kind of taxonomy of bug patterns can also be found. The authors report results of analysis of concurrency errors in several real-life programs. A detailed description of all possible concurrency errors that can occur when the `synchronized` construct is used in Java is provided in [71] where a Petri net model of this synchronisation construct is analysed. In comparison to these works, our aim is to provide uniform definitions of common concurrency errors that are not based on some specific set of programs or some specific synchronisation means, and the stress is always laid on the generic skeleton of the definitions and the notions parameterising it. There are not particular bug patterns relied on because these are always incomplete, characterising only some specific ways how a certain type of error can arise.

Errors including concurrency errors can be divided according to the type of correctness property they violate into two groups: (i) *Safety errors* violate safety properties of a program that must *always* be true, i.e., their violation cause something bad to happen. They always have a finite witness leading to an error state. (ii) *Liveness errors* are errors which violate liveness properties of a program that must *eventually* become true, i.e., their violation prevent something good from happening. They have infinite (or finite but complete—i.e., not further extendable) witnesses. In the rest of this section, first define safety and then liveness errors are defined.

2.2.1 Safety Errors

Safety errors are easier to detect and debug due to their finite witnesses and therefore are more studied in the papers. This subsection describes five concurrency errors, namely: data races, atomicity violations, order violations, deadlocks, and missed signals.

Data races Data races are one of the most common (mostly) undesirable phenomena in concurrent programs. To be able to identify an occurrence of a data race in an execution of a concurrent program, one needs to be able to say (i) which variables are shared by any two given threads and (ii) whether any given two accesses to a given shared variable are synchronised in some way. A data race can then be defined as follows.

Definition 1 *A program execution contains a data race iff it contains two unsynchronised accesses to a shared variable and at least one of them is a write access.*

There exists work, c.f., e.g., [87], which claim that some data races are intentional and can be tolerated. Such data races are referred to as *benign* data races while data races which can cause a program failure are called *harmful* data races. Note, that memory models of higher level programming languages like Java expects programs to be data race free in order to reason about correctness. If a program is not data race free, the program may suffer from unexpected behaviour because the memory model does not dictate how the JVM should behave in such situation.

Atomicity Violations Atomicity is a non-interference property. It is rather generic notion parametrised by (i) a specification of when two program executions may be considered equivalent from the point of view of their overall impact and (ii) a specification of which code blocks are assumed to be atomic. Then an atomicity violation can be defined as follows.

Definition 2 *A program execution violates atomicity iff it is not equivalent to any other execution in which all code blocks which are assumed to be atomic are executed serially.*

An execution that violates atomicity of some code blocks is often denoted as an *unserialisable* execution. The precise meaning of unserialisability of course depends on the employed notion of equivalence of program executions—an overview of various notions of such equivalences can be found in Section 2.4 within the discussion of dynamic techniques used for detecting atomicity violations.

As for the blocks to be assumed to be executed atomically, some authors expect the programmers to annotate their code to delimit such code blocks [41]. Some other works come with predefined patterns of code which should be typically executed atomically [75, 109, 47]. Still other authors try to infer blocks to be executed atomically, e.g., by analysing data and control dependencies between program statements [115], where dependent program statements form a block which should be executed atomically, or by finding out access correlations between shared variables [73], where a set of accesses to correlated shared variables should be executed atomically (together with all statements between them).

Order Violations Order violations form a much less studied class of concurrency errors than data races and atomicity violations. An order violation is a problem of a missing enforcement of some *higher-level* ordering requirements (e.g., a file must be opened before use). For detecting order violations, one needs to be able to decide for a given execution whether the instructions executed in it have been executed in the right order. An order violation can be defined as follows.

Definition 3 *A program execution exhibits an order violation if some instructions executed in it are not executed in an expected order.*

Deadlocks Deadlocks represent a class of safety errors which is quite often studied in the literature. However, despite that, the understanding of deadlocks still varies in different works. We stick here to the meaning common, e.g., in the classical literature on operating systems, c.f., e.g., [105]. To define deadlocks in a general way, it is assumed that, given any state of a program, (i) one can identify threads that are blocked and waiting for some event to happen, and (ii) for any waiting thread t, one can identify threads that could generate an event that would unblock t.

Definition 4 *A program state contains a set S of deadlocked threads iff each thread in S is blocked and waiting for some event that could unblock it, but such an event could only be generated by a thread from S.*

Most works consider a special case of deadlocks—namely, the so-called *Coffman deadlock* [28]. A Coffman deadlock happens in a state in which four conditions are met: (i) Processes have an exclusive access to the resources granted to them, (ii) processes hold some resources and are waiting for additional resources, (iii) resources cannot be forcibly removed from the tasks holding them until the resources are used to completion (no preemption on the resources), and (iv) a circular chain of tasks exists in which each task holds one or more resources that are being requested by the next task in the chain. Such a definition perfectly fits deadlocks caused by blocking lock operations but does not cover deadlocks caused by message passing (e.g., a thread t_1 can wait for a message that could only be sent by a thread t_2, but t_2 is waiting for a message that could only be sent by t_1).

Missed Signals Missed signals are another less studied class of concurrency errors. The notion of missed signals assumes that it is known which signal is *intended* to be delivered to which thread or threads. A missed signal error can be defined as follows.

Definition 5 *A program execution contains a missed signal iff there is sent a signal that is not delivered to the thread or threads to which it is intended to be delivered.*

Since signals are often used to unblock waiting threads, a missed signal error typically leads to a thread or threads being blocked forever.

2.2.2 Liveness and Mixed Errors

Dealing with liveness errors is much harder than with safety errors because algorithms dealing with them have to find out that there is no way something could (or could not) happen in the future, which often boils down to a necessity of detecting loops. Mixed errors are then errors that have both finite witnesses as well as infinite ones whose any finite prefix does not suffice as a witness.

Before discussing more concrete notions of liveness and mixed errors, lets first introduce the very general notion of *starvation* [105].

Definition 6 *A program execution exhibits starvation iff there exists a thread which waits (blocked or continually performing some computation) for an event that needs not to occur.*

Starvation can be seen to cover as special cases various safety as well as liveness (or mixed) errors such as deadlocks, missed signals, and the below discussed livelocks or blocked threads. In such cases, an event for which a thread is waiting cannot happen, and such a situation is clearly to be avoided. On the other hand, there are cases where the event for which a thread is waiting can always eventually happen despite there is a possibility that it never happens. Such situations are not welcome since they may cause performance degradation, but they are sometimes tolerated (one expects that if an event can always eventually happen, it will eventually happen in practice).

Livelocks and Non-Progress Behaviour There are again various different definitions of a livelock in the literature. Often, the works consider some kind of a *progress* notion for expressing that a thread is making some useful work, i.e., doing something what the programmer intended to be done. Then they see a livelock as a problem when a thread is not blocked but is not making any progress. However, by analogy with deadlocks, it seems more appropriate to restrict the notion of livelocks to the case when threads are looping in a useless way while trying to synchronise (which is a notion common, e.g., in various works on operating systems). That is why, first a general notion of non-progress behaviour is defined and then it is specialised to livelocks.

Definition 7 *An infinite program execution exhibits a non-progress behaviour iff there is a thread which is continually performing some computation, i.e., it is not blocked, but it is not making any progress.*

A non-progress behaviour is a special case of starvation within an infinite behaviour. On the other hand, starvation may exhibit even in finite behaviours and also in infinite progress behaviours in which a thread is for a while waiting for an event that is not guaranteed to happen.

In the literature there are used various different notions of progress. Some works, e.g., [100], define progress by looking at the *communication* among two or more co-operating threads. If a thread communicates, it is progressing. In [44], progress is associated with operations on the so-called *communication objects* (such as shared variables, semaphores, FIFO buffers, etc.). In [69, 103], progress is defined by reaching a so-called *progress action* or *progress statement*, respectively (e.g., delivering output, responding to the environment, etc.). In [52], progress is expressed by a so-called *liveness signature*, a set of state predicates and temporal rules, specifying which application states determine whether a program is making a progress when they are repeatedly reached.

As has already been said above, livelocks may be seen as a special case of non-progress behaviour [105].

Definition 8 *Within an infinite execution, a set S of threads is in a livelock iff each of the threads in S keeps running forever in some loop in which it is not intended to run forever, but which it could leave only if some thread from S could leave the loop it is running in.*

There are many, often inconsistent, definitions of a livelock. Moreover, many existing works do not distinguish between livelocks and a non-progress behaviour, c.f., e.g., [19, 100]. Other papers [83, 78] take a livelock to be a situation where a task has such a low priority that it does not run (it is not allowed to make any progress) because there are many other, higher priority, tasks which run instead. Such situation is not considered a livelock and not even a non-progress behaviour but a form of starvation. There are even works [9] for which a thread is in a livelock whenever it is executing an infinite loop, regardless of what the program does within the loop. However, there are many reactive programs which run intentionally in an infinite loop, e.g., controllers, operating systems and their components, etc., and it is not appropriate to consider them to be in a livelock.

Blocked Threads A *blocked thread* appears within some execution when a thread is blocked and waiting forever for some event which can unblock it. Like for a deadlock, one must be able to say what the blocking and unblocking operations are. The problem can then be defined as follows.

Definition 9 *A program execution contains a blocked thread iff there is a thread which is waiting for some event to continue, and this event never occurs in the execution.*

An absence of some unblocking event which leaves some thread blocked may have various reasons. A common reason is that a thread, which should have unblocked some other thread, ended unexpectedly, leaving the other thread in a blocked state. In such a case, one often speaks about the so-called *orphaned threads* [36]. Another reason may be that a thread is waiting for a livelocked or deadlocked thread.

All errors described in this section manifest usually only under certain interleaving of actions executed in different threads. The next section describes techniques that can be used to increase the number of interleavings which programmers can spot during testing of their multi-threaded programs.

2.3 Advanced Techniques of Testing Concurrent Programs

A crucial issue when testing concurrent software is to test as many different (and hopefully relevant) interleavings as possible. To achieve that, one can use, e.g., the methodology proposed in [93] where the author advises to run tests on a multiprocessor system and design the test such that the number of active threads is higher than the number of available processors so that at any given time only some threads are running, thus reducing the predictability of interactions between threads. Such testing scenario based on a high number of running threads is also often called *stress testing* [3].

Advanced approaches to test concurrent software, which are described below, achieve a higher diversity of interleavings even with a low effort required from the programmer. The techniques are based on a repeated execution of the same test with the same inputs and on detecting whether an error occurred during the execution. Such detection can be based on a failure detection, assertion checking, or some dynamic detection technique described in the following Section 2.4. During each execution, the advanced techniques try to affect the scheduler with an intention to see interleavings which have not been spot during the previous executions of the test. The number of different interleavings is increased either by injecting of the so-called *noise* into test executions or by enforcing *deterministic scheduling*.

2.3.1 Noise Injection Techniques

Noise injection techniques, c.f., e.g., [31, 101, 16, 36, 34, 107], inject either randomly or based on some heuristics a noise into the test execution. The noise causes a delay in the execution of a selected thread, giving other threads an opportunity to make a progress.

An advantage of the noise injection approach is that the method does not require any modification of the execution environment nor a manual modification of the test. Instead, the tested system is automatically *instrumented*. The instrumentation modifies the target system with additional code [50]. A program can be instrumented on different levels: Source code, intermediate code such as Java byte-code, or binary

[3]The stress testing is a more general term denoting the process of determining the ability of system to maintain a certain level of effectiveness under unfavorable conditions.

level. The instrumentation can be performed either before the execution or on-the-fly during the execution. In the case of noise injection techniques, the instrumentation typically injects calls to a *noise maker* routine into the program code. Threads executing the modified code then enter the noise maker routine that decides—either randomly or based on some heuristics—whether to cause a noise. Notice that already the instrumentation itself introduce some noise into the execution because the thread must execute the code injected by the instrumentation which transfer control to and from the noise maker.

Interleavings obtained by the noise injection technique are all valid as is discussed in [31] as far as the noise maker does not interfere with the synchronisation policy of the tested program, e.g., by performing an operation on a lock used by the tested program. The noise maker can, however, use synchronisation internally to achieve additional communication among the executed threads.

The noise injection techniques must solve two problems: (i) When to cause a noise (*noise placement problem*) and (ii) how to influence the scheduler (*noise seeding problem*). The first problem includes selection of places where to put instrumentation and decision whether to cause a noise. There is no reason to instrument a code that does not influence synchronisation or communication among threads. Therefore, the noise makers select only concurrency-related events such as execution of routines influencing synchronisation (e.g., locking operations) and accesses to the shared memory [31, 101]. The decision whether to cause a noise at a selected place is done either at random or based on an additional knowledge, e.g., previously detected code pattern, coverage information, or information gathered at run-time [101, 36, 107]. The noise placement problem is discussed in more detail in Chapter 4 where a new noise injection technique is proposed.

The way how the scheduler can be influenced depends on the possibilities available in the run-time environment although the basic principles remains the same. This thesis focuses mainly on Java and therefore possibilities of affecting the Java scheduling are discussed below. However, the described principles can be easily used in other languages and run-time environments.

Noise generation techniques for Java. There exist several ways how a scheduler decision can be affected in Java. In [31], three different *noise seeding* techniques are introduced and evaluated on a single-core processor. The *priority* technique changes priorities of threads. This technique did not provide good results [31]. The *yield* technique injects one or more calls of `yield()` which causes a context switch. The *sleep* technique injects one call of `sleep()`. Experiments presented in [31] showed that the sleep technique provided best results in all cases. However, when many threads were running, the yield technique was also effective. Technique based on sleeps and yields are implemented in the IBM Concurrency Testing Tool (ConTest) [30, 90] and the rstest tool [101].

The current version of IBM ConTest which represents the state of the art in Java noise injection comes with several more noise seeding techniques. The *wait* technique injects a call of `wait()`. The concerned threads must first obtain a special shared monitor, then call `wait()` with a timeout on it, and finally release the monitor. The *synchYield* technique combines the *yield* technique with obtaining the monitor as in the *wait* technique. The *busyWait* technique does not obtain a monitor but instead loops for some time. The *haltOneThread* technique described in [107] occasionally stops one thread until any other thread cannot run. Finally, the *timeoutTamper* heuristics randomly reduces the time-out used when calling `sleep()` in the tested program to test that it is not used for synchronisation[4]. All the above mentioned seeding techniques except the priority technique are parameterised by the so-called *strength of noise*. In the case of techniques based on *sleep* and *wait*, the strength gives the time to wait. In the case of *yield* and *busyWait*, the strength says how many times the yield or the loop implementing busyWait, respectively, should be called.

The noise injection technique is relatively simple, cheap, and effective. The technique is mature enough to be used for testing of real-life software as is the case of IBM ConTest and, for instance, the Microsoft Driver Verifier where the technique is called Concurrency Stress Testing [1].

The noise injection technique can also be used for debugging and healing of previously detected errors. For instance, in [57], a noise is injected into code with intention to achieve an interleaving leading to a situation where a previously detected deadlock manifests. This way, the proposed algorithm checks whether a detected error is real. In [89] and our previous works [58, 67, 61], the noise injection technique was successfully used to prevent previously detected deadlocks, data races, and atomicity violations to manifest during the execution of the code that still contains them.

Still, the noise injection techniques can only influence the scheduler but are not able to effectively control its behaviour to achieve some concrete interleaving. For this purpose, the technique based on deterministic scheduling described next can be used.

2.3.2 Deterministic Testing

Deterministic testing [44, 66, 86, 111] controls thread scheduling decisions during the test execution and systematically explores the interleaving space. Such tools are inspired by the work on model checking introduced in Chapter 1 and can be seen as light-weight model checking (or execution-based model checking). Indeed, the first deterministic testing tools, e.g., [44], use a modified scheduler which handles the *global state* of the test execution and systematically chooses among available

[4]Practical experiences show that the timeoutTamper technique often leads to false alarms when programmers use the `sleep()` routine for other than synchronisation purposes. Therefore, the timeout-Tamper technique was recently removed from IBM ConTest. (Shmuel Ur, personal communication).

state transitions inferred from the state of threads present in the program. The tools explore alternative scheduling scenarios using re-initialisation of the tested program. The state space explosion problem was suppressed by pruning techniques known as *partial-order reduction* methods [43]. Despite this optimisation, the state space of complex multi-threaded programs remained still huge.

To avoid undesirable modification of the execution environment, modern tools for deterministic testing, e.g., [66, 86, 111], focus on application programming interfaces providing synchronisation functionality to the tested programs. The principle is similar to the code instrumentation described above. The calls of synchronisation routines of the run-time environment or OS are intercepted and passed to the *deterministic scheduler*. This scheduler is able to stop threads which should not proceed. The system scheduler therefore schedules only threads allowed to run by the deterministic scheduler. The deterministic scheduler resumes stopped threads when needed.

The common way these techniques work is as follows. First, the tested program is executed without any interference with the deterministic scheduler, and the executed synchronisation is *recorded*. Then, a *search* algorithm is used to infer a scheduling scenario to be forced by the deterministic scheduler during the next execution of the test. The search algorithm is the most complicated part of the deterministic testing tools [86] because the selected scenario must be *reachable* in the tested program and should lead to a synchronisation scenario that is not equivalent to any already observed synchronisation scenarios. The search algorithm is also able to detect completeness of the testing process—a situation when all reachable synchronisation scenarios have already been tested. In such situation, the verification process is finished.

When the next scheduling scenario is inferred by the search algorithm, the tested program is executed again. Firstly, the deterministic scheduler is used to *replay* the selected scenario. At some point, the deterministic scheduler performs a choice that was not performed in any previous execution and drives the execution into a yet unexplored synchronisation scenario. At this point, the deterministic scheduler allow all threads to run. The rest of the execution is recorded and the representation of explored synchronisation scenarios is updated. Then, the search algorithm is executed again.

None of the modern deterministic testing techniques computes the global state of the system. Instead, they compute some representation of the happens-before relation observed during test runs. Each technique uses a different representation suitable for the search algorithm they propose. The search algorithm is used to infer scheduling scenarios to examine during deterministic testing. In [66], the authors propose the so-called *SYN-sequences*. SYN-sequences are totally or partially ordered sequences of synchronisation events executed on synchronisation objects [24].

The CHESS tool [86] computes an *annotated partially-ordered happens-before graph* inspired by the Lamport's happens-before graph [64]. Nodes in this graph represent synchronisation events executed by threads. Edges in the graph form a partial order of their execution. Each node is annotated with a triple (*task*, *syncVar*, *oper*) where the *task* represents a thread (and other schedulable entities, e.g., a timer) that executes the *oper* operation over the *syncVar* synchronisation object. Size of the graph is reduced using the partial-order reduction [43]. Moreover, the tool uses a further optimisation based on a limitation of possible context switches allowed to be performed by the scheduler [85].

In the dBug tool [111], the search algorithm works over a structure called a *decision tree* where nodes correspond to the so-called *decision points*. Edges correspond to intercepted calls. The decision point is defined as a situation where the executed program can make no progress without the deterministic scheduler resuming execution of some of the suspended threads.

Despite the introduced techniques are able to handle large programs thanks to various optimisations, they still suffer from certain limitations. First, they are sensitive to other sources of non-determinism (e.g., input/output events) which make it difficult to replay an already captured scenario. Second, during the replay phase, they usually allow to run only one thread which has a large impact on the performance of the tested program.

In comparison with the noise injection techniques, techniques based on deterministic scheduling are able to achieve a higher coverage of the synchronisation scenarios in small and middle size programs thanks to carefully chosen test scheduling scenarios. These techniques also make debugging much easier because they are able to provide the interleaving scenario that leads to an error and allow programmer to replay this scenario. Therefore, from our point of view, modern deterministic testing techniques are better for debugging and testing of isolated modules for which unit tests exist while noise injection techniques still provide good results for testing of complex systems.

Both the noise injection technique and the technique based on deterministic scheduling are often combined with various dynamic analyses in order to detect a potential for concurrency errors in the examined executions even when an error is not really witnessed. An overview of the detection techniques is provided in the next section.

2.4 Dynamic Analysis of Concurrent Programs

Dynamic analysis already mentioned in Chapter 1 is very popular in verification of concurrent software because reasoning about only one examined execution path is much easier than reasoning about all possible paths and all possible interactions among threads. The enormous number of potential interleavings is in this case naturally reduced by the examined test execution. Of course, dynamic analysis can also be applied on multiple runs. The noise injection and deterministic testing techniques can be then used to increase the number of different interleavings examined by the dynamic analysis.

Dynamic analysis similarly to noise injection techniques can be based on instrumentation. In this case, calls to routines performing the dynamic analysis are injected to chosen places in the code. Further information (e.g., the identification of a variable which is the current thread about to access) can be passed to the detection algorithm as a parameter of such a routine call. Execution of the routines takes resources. Therefore, dynamic analysis influences performance of the application. In terms of the noise injection, the dynamic analysis also represents a source of a certain noise influencing the thread scheduling.

Plenty of research papers presenting a new approach, framework, or tool for dynamic analysis of concurrent programs are published every year. In this section, an overview of available dynamic techniques is provided. A systematic review of more techniques can be found, e.g., in our technical report [37]. In this section, a special attention is put on analyses of data races, atomicity violations, and deadlocks because some of these analyses inspired our work on coverage metrics described in Chapter 3. The following subsections describe techniques for detection of safety and liveness errors as they were defined in Section 2.2.

2.4.1 Detection of Safety Errors

Similarly to taxonomy in Section 2.2, techniques in this section are divided into five groups according to concurrency errors they detect: (i) data races, (ii) atomicity violations, (iii) order violation, (iv) deadlocks, and (v) missed signals. Moreover, techniques of data race detection are divided into three subcategories according to approach they use, and techniques for detection atomicity violations into two subcategories according to the type of atomicity they consider.

Detection of Data Races

Data races are a well-studied concurrency problem. Therefore, there exist many different techniques for their detection. Dynamic techniques which analyse one particular execution of a program are usually based on computing the so-called locksets and/or happens-before relations along the witnessed execution.

Techniques based on *locksets* [96] build on the idea that all accesses to a shared variable should be guarded by a lock. The lockset is defined as a set of locks that guard all accesses to a given variable. Detectors then use an observation that if the lockset associated with a certain shared variable is non-empty, then there is at least one lock such that every access to the shared variable from any thread is protected by this lock, and hence there is no possibility of simultaneous accesses, and so a data race is not possible. The happens-before-based techniques exploit the *happens-before relation* [64] which is defined as the least strict partial order that includes every pair of causally ordered events. Detectors build (or approximate) the happens-before relation among accesses to shared variables and check that no two accesses (out of which at least one is for writing) can happen simultaneously, i.e., without the happens-before relation between them. In Section 2.1, we have already described how the happens-before relation can be built for Java programs.

In the following paragraphs, some of the lockset-based techniques are mentioned followed by selected happens-before-based detection techniques. Finally, some of the techniques which combine both of these approaches are quoted.

Lockset-based algorithms. The first algorithm which used the idea of locksets was Eraser [96]. The algorithm maintains for each shared variable v, a set $C(v)$ of candidate locks for v. When a new variable is initialised, its candidate set $C(v)$ contains all possible locks. Eraser updates $C(v)$ by intersecting $C(v)$ and the set $L(t)$ of locks held by the current thread whenever v is accessed. Eraser warns about a data race if $C(v)$ becomes empty for some shared variable v along the execution being analysed. In order to reduce the number of false alarms, Eraser introduces an internal state $s(v)$ for each shared variable v used to identify whether v is used exclusively by one thread, v is read by multiple threads, or multiple threads change the value of v. The lockset $C(v)$ is then modified only when the variable is shared. A data race is reported only if $C(v)$ becomes empty and v is in the state denoting that the variable is shared among multiple threads and at least one of them access the variable for writing.

The original Eraser algorithm designed for C programs was later modified for programs written in object-oriented languages, c.f., e.g., [110, 26, 20, 118]. The main modification (usually called as the *ownership model*) is inspired by the common idiom used in object-oriented programs where a creator of an object is actually not the owner of the object. Then one should take into account that the creator always accesses the object first, and no explicit synchronisation with the owner is needed because the synchronisation is implicitly arranged by the Java virtual machine. This idea is reflected by inserting a new internal state of the shared variables. The modification introduces a small possibility of having false negatives [58, 110] but greatly reduces the number of false alarms caused by this object-oriented programming idiom.

A problem of techniques based on locksets is that they do not support other synchronisation than locks and therefore produce too many false alarms when applied to common concurrent software.

Happens-before-based algorithms. Most of the happens-before-based algorithms use the so-called *vector clocks* introduced in [80]. The idea of vector clocks for a message passing system is as follows. Each thread has a vector of clocks T_{vc} indexed by thread identifiers. One position in T_{vc} represents the own clock of t. The other entries in T_{vc} hold logical timestamps indicating the last event in a remote thread that is known to be in the happens-before relation with the current operation of t.

Vector clocks are partially-ordered in a point-wise manner ($\sqsubseteq$) with an associated join operation ($\sqcup$) which takes the highest value for each element in the particular vectors and the minimal element (0). The vector clocks of threads are managed as follows: (1) Initially, all clocks are set to 0. (2) Each time a thread t sends a message, it sends also its T_{vc}, and then t increments its own logical clock in its T_{vc} by one. (3) Each time a thread receives a message, it increments its own logical clock by one and further updates its T_{vc} according to the received vector T'_{vc} to $T_{vc} = T_{vc} \sqcup T'_{vc}$.

Algorithms [95] detect data races in systems with locks via maintaining a vector clock C_t for each thread t (corresponding to T_{vc} in the original terminology above), a vector clock L_m for each lock m, and two vector clocks for write and read operations for each shared variable x (denoted W_x and R_x, respectively). W_x and R_x simply maintain a copy of C_t of the last thread that accessed x for writing or reading, respectively. A read from x by a thread is race-free if $W_x \sqsubseteq C_t$ (it happens after the last write of each thread). A write to x by a thread is race-free if $W_x \sqsubseteq C_t$ and $R_x \sqsubseteq C_t$ (it happens after all accesses to the variable).

Maintaining such a big number of vector clocks as described above generates a considerable overhead. Therefore, in [40], the vector clocks of variables from above were mostly replaced by the so-called *epochs* associated with each variable v that are represented as tuples (t,c) where t identifies the thread that last accessed v for write and c represents the value of its clock. The idea behind this optimisation is that, in most cases, a data race occurs between two subsequent accesses to a variable. In such cases, epochs are sufficient to detect unsynchronised write-write and write-read accesses. However, in cases where a write operation needs to be synchronised with multiple preceding read operations (read-write accesses), epochs are not sufficient, and the algorithm has to build an analogy of vector clocks for sequences of read operations.

Hybrid algorithms. An advantage of the vector clock-based algorithms is their precision. However, the big cost of these algorithms inspired many researches to come up with some combination of happens-before-based and lockset-based algorithms. These combinations are often called *hybrid algorithms*, c.f., e.g., [92, 32, 118, 39].

In RaceTrack [118], a notion of a *threadset* was introduced. The threadset is maintained for each shared variable and contains information concerning threads currently working with the variable. The method works as follows. Each time a thread accesses a shared variable, it forms a label consisting of the thread identifier and its current private clock value. The label is then added to the threadset of the variable. The thread also uses its vector clock to identify and remove from the threadset labels that correspond to accesses that are ordered before the current access. Hence the threadset contains solely labels for accesses that are concurrent. At the same time, locksets are used to track locking of variables, which is not tracked by the used approximation of the happens-before relation. Intersections on locksets are applied if the approximated happens-before relation is not able to assure an ordered access to shared variables. If an ordered access to a shared variable is assured by the approximated happens-before relation, the lockset of the variable is reset to the lockset of the thread that currently accesses it.

One of the most advanced lockset-based algorithms that also uses the happens-before relation is Goldilocks presented in [32]. The main idea of this algorithm is that locksets can contain not only locks but also volatile variables and, most importantly, also threads. An appearance of a thread t in a lockset of a shared variable means that t is properly synchronised for using the given variable. The information about threads synchronised for using certain variables is then used to maintain the transitive closure of the happens-before relation via the locksets. An advantage of Goldilocks is that it allows locksets to grow during a computation when the happens-before relation is established between operations over v.

The basic Goldilocks algorithm is relatively expensive but can be optimised by *short circuiting the lockset computation* (three cheap checks, success of any of them is sufficient for race freedom between the two last accesses on a variable) and using *a lazy computation of the locksets* (the locksets are computed only if the previous optimisation is not able to detect that some events are in the happens-before relation). The optimised algorithm has a considerably lower overhead approaching in some cases pure lockset-based algorithms.

While data race detection techniques focus on a conflict between two accesses to the shared memory only, the techniques described below focus on detection of conflicts among more accesses to the shared memory.

Detection of Atomicity Violation

Taking into account the generic notion of atomicity, methods of detecting atomicity violation can be classified according to: (i) The way they obtain information about which code blocks should, in fact, be expected to execute atomically. (ii) The notion of equivalence of executions used (we will get to several commonly used equivalences in the following). (iii) The number of considered shared variables.

Below, first there are discussed approaches of detecting atomicity violations when considering accesses to a single shared variable only and then those which consider accesses to several shared variables.

Atomicity over one variable. Most of the algorithms for detecting atomicity violations are only able to detect atomicity violations within accesses to a single shared variable. They mostly try to detect a situation where two accesses to a shared variable should be executed atomically, but are interleaved by an access from another thread.

In [115], blocks of instructions which are assumed to execute atomically are approximated by the so-called *computational units* (CUs). CUs are inferred automatically from a single program trace by analysing data and control dependencies between instructions. First, a dependency graph is created which contains control and read-after-write dependencies between all instructions. Then the algorithm tries to partition this dependency graph to obtain a set of distinct subgraphs which are the CUs. The partitioning works in such a way that each CU is the largest group of instructions where all instructions are control or read-after-write dependent, but no instructions which access shared variables are read-after-write dependent, i.e., no read-after-write dependencies are allowed between shared variables in the same computational unit. Since these conditions are not sufficient to partition the dependency graph to distinct subgraphs, additional heuristics are used. Atomicity violations are then detected by checking if the strict 2-phase locking (2PL) discipline [33] is violated in a program trace. Violating the strict 2PL discipline means that some CU has written or read a shared variable which another CU is currently reading from or writing to, respectively (i.e., some CU accessed a shared variable and before its execution is finished, another CU accesses this shared variables). If the strict 2PL discipline is violated, the program trace is not identical to any serial execution, and so seen as violating atomicity. Checking if the strict 2PL discipline is violated is done dynamically during a program execution in case of the online version of the algorithm, or a program trace is first recorded and then analysed using the off-line version of the algorithm.

A simpler approach of discovering atomicity violations was presented in [75]. Here, any two consecutive accesses from one thread to the same shared variable are considered an atomic section, i.e., a block which should be executed atomically. Such blocks can be categorised into four classes according to the types of the two accesses (read or write) to the shared variable. Serialisability is then defined based on an analysis of what can happen when a block b of each of the possible classes is interleaved with some read or write access from another thread to the same shared variable as the one accessed in b. Out of the eight total cases arising in this way, four (namely, r/w/r, w/w/r, w/r/w, r/w/w) are considered to lead to an unserialisable execution. However, the detection algorithm does not consider all the unserialisable executions as errors. Detection of atomicity violations is done dynamically in two steps. First, the algorithm analyses a set of correct (training) runs in which it tries to detect atomic sections which are never unserialisably interleaved. These atomic sections are called *access interleaving invariants* (AI invariants). Then the algorithm checks if any of the obtained AI invariants is violated in a monitored run, i.e., if there is an AI invariant which is unserialisably interleaved by an access from another thread to a shared variable which the AI invariant (atomic section) accesses. While the second step of checking AI invariants violation is really simple and can be done in a quite efficient way, the training step to get the AI invariants can lead to a considerable slow down of the monitored application.

A more complicated approach was introduced in [39, 112] where atomicity violations are sought using the Lipton's reduction theorem [70]. The approach is in particular based on checking whether a given run can be transformed (reduced) to a serial one using commutativity of certain instructions (or, in other words, by moving certain instructions left or right in the execution). Both [39] and [112] use procedures as atomic blocks by default, but users can annotate blocks of code which they assume to execute atomically to provide a more precise specification of atomic sections for the algorithm. For the reduction used to detect atomicity violations, all instructions are classified, according to their commutativity properties, into 4 groups: (1) *Right-mover* instructions R which can be swapped with immediately following instructions. (2) *Left-mover* instructions L which can be swapped with immediately preceding instructions. (3) *Both-mover* instructions B which can be swapped with preceding or following instructions. (4) *Non-mover* instructions N which are not known to be left or right mover instructions. Classification of instructions to these classes is based on their relation to synchronisation operations, e.g., lock acquire instructions are right-movers, lock release instructions are left-movers, and race free accesses to variables are both-movers (a lockset-based dynamic detection algorithm is used for checking race freeness). An execution is then serialisable if it is deadlock-free and each atomic section in this execution can be reduced to a form $R^*N^?L^*$ by moving the instructions in the execution in the allowed directions. Here, $N^?$ represents a single or no non-mover instruction. Both-mover instructions B can be taken as right-mover instructions R or left-mover instructions L.

Atomicity over multiple variables. The already described algorithms consider atomicity of multiple accesses to the same variable only. However, there are situations where we need to check atomicity over multiple variables, e.g., when a program modifies three different variables representing a point in a three-dimensional space. Even if it is ensured that every consecutive pair of read and write accesses to each of these variables is executed atomically, the program can still have an unserialisable execution. This is because the three atomic blocks guarding each pair of accesses to each of these variables can be interleaved with other atomic blocks operating with these variables. Some of these variables can then end up modified by a different thread than the others which cannot happen in a serial execution. The above discussed detectors would not detect any atomicity violation here.

In [11], the problem of violation of atomicity of operations over multiple variables is referred to as a *high-level data race*. In the work, all synchronised blocks (i.e., blocks of code guarded by the `synchronised` statement) are considered to form atomic sections. The proposed detection of atomicity violations is based on checking the so-called *view consistency*. For each thread, a set of views is generated. A view is a set of fields (variables) which are accessed by a thread within a single synchronised block. From this set of views, a set of maximal views (maximal according to set inclusion) is computed for the thread. An execution is then serialisable if each thread is only using views which are compatible, i.e., form a chain according to set inclusion, with all maximal views of other threads. Since the algorithm has to operate with a big number of sets (each view is a set), it suffers from a big overhead.

A different approach is associated with the Velodrome detector [41]. Here, atomic sections (called transactions) are given as methods annotated by the user. Detection of atomicity violations is based on constructing a graph of the *transactional happens-before relation* (the happens-before relation among transactions). An execution is serialisable if the graph does not contain a cycle. The detection algorithm uses a dynamic analysis to create the graph from a program trace and then checks if it contains a cycle. If so, the program contains an atomicity violation. Since creating the graph for an entire execution is inconvenient, nodes that cannot be involved in a cycle are garbage collected or not created at all. Like the previous algorithm, Velodrom too may suffer from a considerable overhead in some cases.

The simple idea of *AI invariants* described above has been generalised for checking atomicity over pairs of variables [109, 47] too. In [109], an algorithm which infers the so-called *units of work* (w.r.t. the set of accesses to the shared memory that must be executed together) is proposed. This is done statically using a dataflow analysis. An execution is then considered serialisable if it does not correspond to any of the problematic interleavings of the detected units of work. An algorithm capable of checking unserialisability of execution of units of work (called atomic-set-serialisability violations) is described in [47], based on a dynamic analysis of program traces. The algorithm introduces the so-called *race automata* which are simple finite state automata used to detect the problematic interleaving scenarios.

Both data race and atomicity violation detection techniques focus primarily on conflicts among accesses to the shared memory. The following detection techniques focus on a problem when two (or more) accesses are not in conflict but their execution in a wrong order leads to a failure.

Detection of Order Violations

Like in the case of atomicity violations, a prerequisite for detecting order violations is to know which order restrictions are assumed. These can be specified manually, generic order requirements may be used (e.g., an object must be first initialised and only then used), or some restrictions may be automatically inferred. The order restrictions considered in current approaches are often quite simple, frequently considering only pairs of instructions. Indeed, the current lack of works targeting high-level order has been pinpointed, e.g., in [74]. The following works focus mainly on order violation detection on the level of accesses to the shared memory—the same level as previously mentioned techniques for detection of data races and atomicity violations.

In [117], authors introduce, for each memory operation o, a set of memory operations $PSet(o)$ which o depends upon and which can safely occur before o. These sets are extracted from a set of correct executions of the analysed program. Then, order violations are sought in further runs by looking for a memory operation o such that the previous memory operation dependent upon o is not in $PSet(o)$.

The ConMem tool [119] detects several behavioural patterns corresponding to order violations that can lead to a program crash. For each test input, ConMem monitors one execution of the given program. It uses a dynamic analysis to first identify parts of executions (denoted as ingredients) that may lead to a crash if ordered differently than in the given execution (e.g., assignments of `null` to a shared pointer and dereferences of this shared pointer from different treads). Then, ConMem analyses synchronisation in the witnessed traces around these potentially problematic constructions to see whether fatal interleavings exist to trigger an error (e.g., an interleaving where a thread t_1 assigns `null` to a shared variable v, and subsequently, a thread t_2 dereferences v). The paper describes four problematic patterns consisting of ingredients and timing conditions that lead to an error and that ConMem is able to detect. One example is the *Con-NULL* pattern with ingredients rp—a thread t_1 reads a pointer ptr, wp—a thread t_2 writes `null` to ptr, and timing conditions requiring wp to execute before rp with no write operation on ptr happening in between of wp and rp. Another example is the *Con-UnInit* pattern with ingredients r—a thread t_1 reads a variable v without previously writing to v, w—a thread t_2 initialises v, and the timing condition requiring r to execute before w.

All so-far described detection techniques look for possibly erroneous accesses to shared memory. The following techniques focus on violations in use of synchronisation mechanisms that usually cause a program to hang instead of crash. We first focus on problems with locks and then with the wait-signal synchronisation mechanism.

Detection of Deadlocks

Detection of deadlocks usually involves various graph algorithms as it is, for instance, in the case of the algorithm introduced in [91] where a *thread-wait-for graph* is dynamically constructed and analysed for a presence of cycles. Here, a thread-wait-for graph is an arc-labelled digraph $G = (V, E)$ where vertices V are threads and locks, and edges E represent waiting arcs which are classified (labelled) according to the synchronisation mechanism used (join synchronisation, notification, finalisation, and waiting on a monitor). A cycle in this graph involving at least two threads represents a deadlock. A disadvantage of this algorithm is that it is able to detect only deadlocks that actually happen. The following works can detect also potential deadlocks that could happen but did not actually happen during the witnessed execution.

In [49], a different algorithm called GoodLock for detecting deadlocks was presented. The algorithm constructs the so-called *runtime lock trees* and uses a depth-first search to detect cycles in it. Here, a runtime lock tree $T_t = (V, E)$ for a thread t is a tree where vertices V are locks acquired by t, and there is an edge from $v_1 \in V$

to $v_2 \in V$ when v_1 represents the most recently acquired lock that t holds when acquiring v_2. A path in such a tree represents a nested use of locks. When a program terminates, the algorithm analyses lock trees for each pair of threads. The algorithm issues a warning about a possible deadlock if the order of obtaining the same locks (i.e., their nesting) in two analysed trees differs, and no "gate" lock guarding this inconsistency has been detected.

The original GoodLock algorithm is able to detect deadlocks between two threads only. Later works, e.g., [17, 4] improve the algorithm to detect deadlocks among multiple threads. In [4], a support for semaphores and wait-notify synchronisation was added. The recent work [57] modified the original algorithm so that runtime lock trees are not constructed. Instead, the algorithm uses a stack to handle the so-called *lock dependency relation*. The algorithm computes the transitive closure of the lock dependency relation instead of performing a depth-first search in a graph. The modified algorithm uses more memory but the computation is much faster.

Detection of Missed Signals

There are not many works focusing specially on missed signals. Usually, the problem is studied as a part of detecting other concurrency problems. In [4], a lost notification error is reported if there is a notify event e in a trace tr and there exists a trace that is a *feasible permutation* of tr in which e wakes up fewer threads than it does in tr. Such a situation is possible when the wait event of one of the threads woken in tr is not constrained to happen before the event e.

2.4.2 Detection of Liveness Errors

As in the previous subsection, techniques for detection of liveness errors described below are divided according to the type of errors they detect: (i) livelocks and non-progress behaviour and (ii) blocked threads.

Detection of Livelocks and Non-progress Behaviour

We are not aware of any works specialising in detection of livelocks in the sense we defined them in Section 2.2, which require not only detection of a looping behaviour but also of the fact that this behaviour could be escaped only if some of the livelocked threads could escape it. There are, however, works considering detection of non-progress behaviour (sometimes under the name of livelock detection, but we stick here to speaking about detection of non-progress behaviour).

To enable a non-progress detection, the general notions of progress discussed in Section 2.2 have to be concretised for a particular program. This is mostly expected to be done by the user, e.g., by specifying progress actions [69], labelling statements as progress statements [53], annotating the code [52], etc.

There are only few approaches which use dynamic analysis for detection of non-progress behaviour. In [52], a liveness signature, a set of state predicates and temporal rules, is used to determine whether a program is making progress. In other words, the liveness signature determines which program states are significant in determining whether a program is making a progress. If a program does not reach any of these program states for some time, the program is at a so-called *standstill*, i.e., it is not making progress.

Detection of Blocked Threads

We are not aware of any works specialising in this kind of errors. Of course, the most simple solution to deal with this error is to check that a thread is waiting for more than some time to be unblocked. This is, however, a very crude approach. In fact, a similar approach is used in MySQL to detect deadlocks. It was shown [74] that it is quite inaccurate detection method which often leads to false alarms and, in case of MySQL, to unnecessary restarts.

Chapter 3

Concurrency Coverage Metrics

In testing, testers need measures that can be used to asses how well a program has been tested, how good a test is, or whether further testing is necessary. For this purpose, the concept of *coverage metrics* introduced in Chapter 1 is used. Recall that coverage metrics are based on *coverage tasks* representing different phenomena whose occurrence in the behaviour of a tested program is considered to be of interest.

Probably the most popular measure is the *code coverage* [94] which measures how much of the code (the number of lines, the number of executed statements, the number of branch conditions covered both ways, etc.) has been executed during a test execution. A high code coverage is a necessary condition for a good verification. Moreover, the fact that every line of the code has been executed does not imply that all the functionality and possible behaviour has been exercised. Therefore, besides code coverage, testers often measure also other kinds of coverage like *functional coverage* capturing the tested functionality scenarios, etc. The *concurrency coverage* metrics discussed in this chapter measure how well the synchronisation mechanisms and various other concurrency-related aspects of the behaviour has been exercised.

A common goal of the testing process is to reach a *full coverage*, i.e., to cover all tasks of the coverage domain. However, obtaining a full coverage for a complex software and nontrivial metrics is often difficult and expensive. Moreover, for many nontrivial metrics, it is very difficult and in general undecidable to statically determine reachable coverage tasks and hence full coverage. For instance, in the case of the relatively trivial code coverage, one has to identify dead code (the code that is not reachable from the entry point of the tested program using any possible input data) in order to be able to identify full coverage. Coverage metrics without a known full coverage can, however, still be used in various ways. First, they can be used for comparisons of testing techniques and tests. Second, they can be used to control termination of the testing process within the so-called *saturation-based testing* [98] where the so-called *saturation effect*, i.e., a situation when the obtained coverage stops growing, can be used to determine whether the testing can be stopped. Finally, they are also useful in *search-based testing* discussed in Chapter 5 where the coverage metrics can be used within an objective function to compare candidate solutions.

For metrics used in saturation-based or search-based testing, one can identify several specific properties that they should exhibit. First, within the testing process, the obtained coverage should grow for a while as often as possible and then stabilise. Hence, it should not immediately jump to some value and stabilise on it. On the other hand, it should not take too much time for the coverage to stabilise. Also, to enable a reliable detection of stabilisation, the coverage should grow as smoothly as possible, i.e., without growing through a series of distinctive shoulders. Next, in case of testing an erroneous program, the stabilisation should ideally not happen before an error is detected. Finally, the increase in coverage should be linked with witnessing more and more behaviours that differ in their potential of exhibiting a bug.

In this chapter, several new coverage metrics suitable for saturation-based or search-based testing of concurrent programs are provided. These metrics are based on coverage tasks derived from the information about program behaviour that is gathered or computed by various dynamic analyses, namely, Eraser [96], GoldiLocks [32], AVIO [75], and GoodLock [17] discussed in Section 2.4. In fact, the idea of inferring new metrics from these analyses is rather generic and can be applied to other dynamic as well as static analyses (even those that will appear in the future) too.

The proposal is motivated by the idea that within the development of such analyses, behavioural aspects of concurrent programs that are highly relevant for the existence of synchronisation-related errors have been identified. Hence, it makes sense to measure how well the aspects of the behaviour tracked by such analyses have been covered during testing.

Further, coverage tasks of the newly proposed as well as some existing metrics are combined with *abstract identifiers of the threads* involved in generating the phenomena reflected in the concerned tasks. The identifiers abstract away the particular numerical identifiers of the threads, but preserve information on their type, the history of their creation, etc. This way, an increased number of coverage tasks is obtained, forming a new, more precise variant of the original metric.

In the rest of this chapter, existing concurrency and synchronisation coverage metrics are discussed. Next, a methodology how metrics satisfying the requirements for saturation-based and search-based testing is provided. Then, several new particular metrics are proposed and some existing metrics mentioned below are described (in one case also extended). Finally, a comparison of the proposed metrics with selected existing metrics on a set of concurrent programs is provided.

3.1 Related Work

In the past decades, many different coverage metrics were proposed targeting probably all areas of testing. Testing of concurrent software is not an exception. The existing concurrency-related metrics can be divided into two groups: (i) The *interleaving-based* metrics simply track the ordering in which a pair (or a set) of events occur, i.e., they measure how many of such different interleavings have appeared. (ii) The *scheduling-based* metrics in addition track which different synchronisation actions have been used among the threads. Apart from that, there has also been proposed a metric called *synchronisation coverage* [22] which does not fit to any of the two groups. The metric focuses on usage scenarios of synchronisation constructs.

The interleaving-based metrics, c.f., e.g., [116, 72, 22] simply track occurrences of certain important events in multiple threads. In [116], the well-known structural path metric based on the all-definition-use pairs (du-pairs) [113] was extended to support parallel programs. A *definition-use pair* is a pair of instructions (d,u) where instruction d writes a value to a variable, and the u instruction subsequently reads the value of the variable. All-du-pairs coverage therefore measures how many of such pairs of write and subsequent read operations appeared in a testing run.

In [72], the interleaving of operations accessing shared memory is studied. The authors discuss seven interleaving-based coverage metrics including the all-du-pairs metric. These coverage metrics do not consider synchronisation operations and focus only on the interleaving among accesses to shared memory. The authors start with a metric measuring all interleavings of all accesses to shared memory. Then, they limit this rather generic metric such that they measure only accesses performed by a pair of threads, accesses to a single variable, any pair of accesses to single shared variable, and finally, pair of accesses performed by different threads. A theoretical comparison of the discussed metrics is provided in the paper.

The *concurrent pairs of events* metric described in [22] considers interleaving scenarios among all pairs of concurrency related operations including accesses to shared variables and operations over synchronisation primitives. Each coverage task consists of a pair of operations that are assumed to be encountered consecutively in a run and an additional boolean tag. It is $false$ iff the two operations are executed by the same thread, and $true$ otherwise (indicating that a context switch occurred).

The scheduling-based metrics, e.g., [106, 84] often work with some graph representation of possible scheduling scenarios. In [106], the authors define a coverage metric over a *concurrency state graph*. Nodes of the graph represent the actual state of program (as a composition of states of individual threads, denoted as *cc-states*), and edges represent synchronisation events generated by the threads. The paper proposes five coverage metrics. Three of them are state-transition-based, namely, *all-concurrency-paths*, *all-proper-cc-histories*, and *all-edges-between-cc-states*. These metrics measure how many of different paths in the concurrency state graph appeared in a testing run. They precisely capture all sequences of synchronisation events but are difficult to be efficiently computed. In the case of *all-concurrency-paths*, the metrics considers even infinite paths. The two other metrics are state-based. The *all-cc-states* metric measures the number of cc-states executed during a test, and the *all-possible-rendezvous* metric counts only those cc-states that indicate that some synchronisation occurred between the threads. With respect to a possibly very large concurrency state graph, the use of these metrics is also impractical.

A different model was used in [84]. The paper models concurrent programs using high-level *Petri-nets* where each test execution is represented as a firing sequence with particular input/output data. The coverage metrics measure how many of markings (no data are considered) or how many of transition firings (input/output data are considered) appeared in a testing run.

A disadvantage of scheduling-based coverage metrics mentioned above is a high computational cost and possibly large number of coverage tasks. Note, however, that the approach of deterministic testing introduced in Chapter 2 is in fact in some sense similar to maximising coverage under a scheduling-based metric. However, instead of enumerating coverage tasks, these techniques compute some optimised representation of possible synchronisation scenarios [66].

Our metrics proposed in this chapter can be considered as interleaving-based coverage metrics or lightweight scheduling-based metrics. With the price similar to interleaving-based metrics our metrics also capture recently performed synchronisation via *context information* inspired by the internal state of the selected detection algorithm. The incorporation of algorithms for dynamic analysis of concurrent programs is novel and leads to metrics which need fewer coverage tasks to capture important aspects of concurrency-related behaviour of tested programs.

Furthermore, the extended versions of our metrics make thread and object identification explicit to the coverage tasks. The idea of extending coverage tasks of metrics by thread identification has also been recently presented in [98]. In this work, the authors propose two types of context information which can be used to refine existing metrics. The *group context* makes explicit the *type* of thread that performed an event and the *thread context* which explicitly identifies the thread which performed the event. These contexts can be seen as special cases of our abstract thread identification.

3.2 Methodology of Deriving New Coverage Metrics

In this section, our methodology for deriving new coverage metrics is provided. To derive metrics satisfying the criteria set up above, we propose to get inspired by various existing *dynamic (and possibly even static) concurrency error detection techniques*. This is motivated by two observations: (i) These detection techniques focus on those events occurring in runs of the analysed programs that appear relevant for detection of various concurrency-related errors. (ii) The techniques build and maintain a representation of the context of such events that is important for detection of possible bugs in the program. Hence, trying to measure how many of such events have been seen, and possibly in how many different contexts, seems promising from the point of view of relating the growth of a metrics to an increasing likelihood of spotting an error.

The described idea is very generic, and one can speak about a new class of concurrency coverage metrics that can be obtained in the described manner. A crucial step in the creation of a new coverage metrics based on some error detection algorithm is to choose suitable pieces of information available to or computed by the detection algorithm, which are then used to construct the domain of the new coverage metrics such that the other, above mentioned criteria are met. This leads to a trade off among the precision of the metrics and the amount of information tracked, the associated computational complexity, and speed of saturation. One extreme is to build a coverage metrics directly on warnings about concurrency errors issued by the detection algorithm. In this case, the detection algorithm entirely need to be implemented. Another extreme is to build a coverage metric counting just the events tracked by the detection algorithm, without their context. In such a case, very similar metrics to already existing metrics are often obtained. Within this process—which can hardly be made algorithmic and which requires certain ingenuity and also experimental evidence, it can also of course turn out that some detection algorithms are not suitable as a basis of a coverage metrics at all.

Let us demonstrate the described problem on an example of two kinds of dynamic data race detection algorithms. The *vector-clock-based algorithms*, e.g., [95] described in Chapter 2, maintain for each thread an internal clock which is an integer value representing the number of synchronisation events that the thread executed so far. Vectors of clocks are not suitable for our purposes because they encode the history context using a too large number of values. In fact, a coverage metric based on vector-clocks would be very similar to metrics based on the concurrency state graph [106] described above because both of the approaches are based on the same model of multi-threaded execution. Such metrics lead to a huge number of coverage tasks, a slow progress towards saturation, and also a high cost of measuring the obtained coverage.

On the other hand, the Eraser algorithm [96] also described in Chapter 2, which computes *locksets*, is more suitable for our purposes because the history context used by it gives rise to a reasonable number of coverage tasks.

Finally, already mentioned above, according to our experimental evidence mentioned later on, the precision of the constructed metrics can further be suitably adjusted by combining their coverage tasks with some *abstract identification of the threads* involved in generating the phenomena reflected in the concerned tasks. The identification should of course not be based on the unique thread identifiers, but it can preserve information on their type, the history of their creation, etc. A similar identification can then also be used whenever the coverage tasks contain some dynamically instantiated objects (e.g., locks). Our approach for constructing thread and object identifiers is described in Section 3.4.

Table 3.1: The considered coverage metrics

metrics	coverage task	note
Avio	(pl_1, pl_2, pl_3)	N
Avio*	$(pl_1, pl_2, pl_3, var, t_1, t_2)$	N
Eraser	$(pl_1, state, lockset)$	N
Eraser*	$(pl_1, var, state, lockset, t_1)$	N
GoldiLock	$(pl_1, goldiLockSetSC)$	N
GoldiLock*	$(pl_1, var, goldiLockSetSC, t_1)$	N
GoodLock	(pl_1, pl_2, l_1, l_2)	N
GoodLock*	$(pl_1, pl_2, l_1, l_2, t_1)$	N
HBPair	$(pl_1, pl_2, syncObj)$	N
HBPair*	$(pl_1, pl_2, syncObj, t_1, t_2)$	N
ConcurPairs	$(pl_1, pl_2, switch)$	E
DUPairs	(pl_1, pl_2, var)	E
DUPairs*	$(pl_1, pl_2, var, t_1, t_2)$	M
Sync	$(pl_1, mode)$	E

3.3 Novel Coverage Metrics

In this section, several new particular coverage metrics are going to be derived. As mentioned above, they are all based on some dynamic analyses used for detecting errors in synchronisation of concurrent programs. In order to allow for a quick comparison among the metrics, Table 3.1 presents an overview of all the proposed metrics, together with some other metrics that we will consider in our experiments. For each metric, the second column shows tuple defining coverage tasks of the metric, and the third column contains information whether the metric is new (N), already existing (E), or whether it is our modification of some already known metric (M). The first item of each of the tuples representing a coverage task (denoted as pl_1) gives a primary program location which generates the given task when reached by some thread. The rest of the tuples can then be viewed as a context under which the location is reached. For most of the metrics, we provide two versions: a basic version and a version with an extended context, denoted by *. In the following paragraphs, the versions with the extended context are described only. The basic versions can easily be derived from them by dropping some elements of the context.

In the text below, the following notation is used. V is a set of identifiers of instances of non-volatile variables (i.e., non-volatile fields of objects) that may be used in the tested program at hand, O is a set of identifiers of instances of volatile variables used in the program, L is a set of identifiers of locks used in the program, T is a set of identifiers of all threads that may be created by the program, and P is a set of all program locations in the program. One possible particular way how the needed identifiers may be obtained is discussed in Section 3.4.

Coverage metrics based on Eraser. The coverage metric Eraser* is based on the Eraser algorithm [96]. Its coverage tasks have the form of a tuple $(pl_1, var, state, lockset, t_1)$ where $pl_1 \in P$ identifies the program location of an instruction accessing a shared variable $var \in V$, $state \in \{virgin, exclusive, exclusive', shared, modified, race\}$ gives the state in which the Eraser's finite control automaton is when the given location is reached (we consider the extended version of Eraser using the *exclusive'* state as introduced in [110], which is more suitable for the Java memory model), and $lockset \subseteq L$ denotes a set of locks currently guarding the variable var. Finally, $t_1 \in T$ represents the thread performing the access operation.

Coverage metrics based on GoldiLocks. GoldiLock* is based on the Goldilocks algorithm [32]. Original algorithm is quite complex. Therefore, we focus on the version with *short circuit checks* (SC) which are three cheap checks that are sufficient for deciding race freedom. We choose this algorithm because it is able to handle happens-before relation using relatively small set of elements in the lockset. Our GoldiLock-based metric GoldiLock* is based on coverage tasks having the form of tuples $(pl_1, var, goldiLockSet, t_1)$ where $pl_1 \in P$ gives the location of an instruction accessing a variable $var \in V$ within a thread $t_1 \in T$, and $goldiLockSetSC \subseteq O \cup L \cup T$ represents the lockset computed by GoldiLocks. If the lockset is not necessary to compute due to the SC, the element used by the SC is used instead. For instance, $goldiLockSetSC = t$ if the SC checking whether the current and the last accesses to the variable are performed by the same thread t succeeded.

Coverage metrics based on Avio. The Avio algorithm that detects atomicity violation over one variable is presented in [75]. We choose this algorithm because it does not require any additional information from the user about instructions that should be executed atomically. The algorithm considers any two consecutive accesses a_1 and a_2 from one thread to a shared variable var to form an atomic block B. Serialisability is then defined based on an analysis of what can happen when B is interleaved with some read or write access a_3 from another thread to the variable var. Tracking of all accesses that occur concurrently to a block B can be very expensive. Therefore, we define our criterion to consider only the last interleaving access to the concerned variable from a different thread. Our Avio* metric uses coverage tasks in the form of tuples $(pl_1, pl_2, pl_3, var, t_1, t_2)$ where $var \in V$, $pl_1, pl_2, pl_3 \in P$, and $t_1, t_2 \in T$. The considered atomic block B spans between pl_1 and pl_2, and it is executed by a thread t_1. Finally, pl_3 gives a location of an instruction executed in a thread t_2 that interferes with the block B.

Coverage metrics based on GoodLock. GoodLock is a popular deadlock detection algorithm that exists in several modifications—we, in particular, build on its modification published in [17]. The algorithm builds the so-called *guarded lock graph* which is a labelled oriented graph where nodes represent locks, and edges represent nested locking within which a thread that already has some lock asks for another one. Labels over edges provide additional information about the thread that creates the edge. The algorithm searches for cycles in the graph wrt. the edge labels in order to detect deadlocks. Our metric focuses on occurrence of nested locking that is considered interesting by GoodLock. We omit collection of the locksets of the threads which the original algorithm uses as one element of the edge label because this information is used in the algorithm to suppress certain false alarms only. Our GoodLock* metric is therefore based on coverage tasks in the form of tuples $(pl_1, pl_2, l_1, l_2, t_1)$ where $pl_1, pl_2 \in P$, $l_1, l_2 \in L$, and $t_1 \in T$. Such task is covered when the thread t_1 has obtained the lock l_1 at pl_1, and now the same thread is obtaining the lock l_2 at pl_2.

Coverage metrics based on happens-before pairs. These coverage metrics are motivated by observations we get from the GoldiLocks algorithm and the vector-clock algorithms, both of them depending on a computation of the happens-before relation. In order to get rid of a possibly huge number of coverage tasks produced by the vector-clock algorithms and trying to decrease the computational complexity needed when the full GoldiLocks algorithm is used, we focus on pieces of the information the algorithms use to create their representations of the analysed program behaviours (without actually computing and using these representations). All of these algorithms rely on synchronisation events constructing the happens-before relation observed along the execution path. Inspired by this, we propose the HBPair* metric that tracks successful synchronisation events based on locks, volatile variables, wait-notify operations, and thread start and join operations used in Java. A coverage task is defined as a tuple $(pl_1, pl_2, syncObj, t_1, t_2)$ where $pl_1 \in P$ is a program location in a thread $t_1 \in T$ that was synchronised with the location $pl_2 \in P$ of the thread $t_2 \in T$ using the synchronisation objects $syncObj \in L \cup O \cup \{\perp\}$. Here, $\perp$ represents a thread start or a successful join synchronisation where no synchronisation object is needed.

In order to compare our metrics with already existing metrics, we further consider and in one case also extend the following metrics.

Coverage based on concurrently executing instructions (ConcurPairs). The interleaving-based coverage of concurrent pairs of events proposed in [22] is a metric in which each coverage task is composed of a pair of program locations that are assumed to be encountered consecutively in a run and a third item that is $true$ or $false$

as was described in Section 3.1. This metric provides statement coverage information (using the $false$ flag) and interleaving information (using the $true$ flag) at once. In our notation, each task of the metric is a tuple $(pl_1, pl_2, switch)$ where $pl_1, pl_2 \in P$ represent the consecutive program locations (only concurrency primitives and variable accesses are monitored), and $switch \in \{true, false\}$ denotes whether the context switch occurs in between of them. Since this metric produces a large number of coverage tasks even for small programs, we decided not to enrich it with any further context information.

Definition-use coverage. This coverage metric is based on the *all-du-path* already explained in Section 3.1. We consider the original all-du-pair coverage metric (denoted as DUPairs), and we also extend it to a metric which adds more context information to the coverage tasks. Our metric DUPairs* is based on coverage tasks in the form of tuples $(pl_1, pl_2, var, t_1, t_2)$ where $pl_1, pl_2 \in P$ represent program locations where the value of the variable $var \in V$ is defined and used, respectively, $t1 \in T$ denotes the thread that performed the definition of var at pl_1, and $t_2 \in T$ denotes the thread that subsequently uses the value at pl_2.

Synchronisation coverage (Sync). The synchronisation coverage [22] focuses on the use of synchronisation primitives and does not directly consider thread interleavings. Coverage tasks of the metric are defined based on various distinctive situations that can occur when using each specific type of synchronisation primitives. For instance, in the case of a synchronised block (defined using the Java keyword `synchronised`), the obtained tasks are: *synchronisation visited*, *synchronisation blocking*, and *synchronisation blocked*. The synchronisation visited task is basically just a code coverage task. The other two are reported when there is an actual contention between synchronised blocks—when a thread t_1 reaches a synchronised block A and stops because another thread t_2 is inside a block B synchronised on the same lock. In this case, A is reported as blocked, and B as blocking (both, in addition, as visited). In our notation, the metric is defined using tuples of the form $(pl_1, mode)$ where $pl_1 \in P$ represents the program location of a synchronisation primitive, and $mode$ represents an element from the set of the distinctive situations relevant for the given type of synchronisation.

3.4 Abstract Object and Thread Identification

Our coverage metrics introduced in the previous section are based on tasks that include identification of threads and instances of variables and locks. The Java virtual machine (JVM) generates identifiers of objects and threads dynamically. Such identifiers are, however, not suitable for our purposes: (i) In long runs, too many of them may be generated. (ii) We would like to be able to match semantically equivalent tasks generated in different runs (may be not precisely, but at least with a reasonable precision), and the identifiers generated by JVM for the same threads (from the semantical point of view) in different runs will quite likely be different.

Previous works, such as [98], used Java types to identify threads. We consider this type-based identification of elements as too rough. Our goal is to create identifiers which distinguish behaviour of objects and threads within the program more accurately, but still keeping a reasonable level of abstraction so the set of such abstract identifiers remains of a moderate size.

The abstract *object identification* that we consider in this work (to identify locks as well as instances of variables[1]) is based on the observation that, usually, objects created in the same place in the program are used in a similar way. For instance, there are usually many instances of the class `String` in an average Java program, but all strings that are created within invocations of the same method will probably be manipulated similarly. Therefore, we define an object identifier as a tuple $(type, loc)$ where *type* refers to the type of the object, and *loc* refers to the top of the stack (excluding calls to constructors) when the object is created. The record at top of the stack contains a method, source file, and line of code.

Next, our abstract *thread identification* is based on an observation that the type and place of creation are not sufficient to build a thread identifier. Several threads created at the same program location (e.g., in a loop) can subsequently process different data and therefore behave differently. We need more information concerning the thread execution trace to better capture the behaviour of threads. Therefore, we use as the identifier a tuple $(type, hash)$ where *type* denotes the type of the object implementing the thread, and *hash* contains a hash value computed over a sequence of n first method identifiers that the thread executed after its creation (if the thread terminates sooner, then all methods it executed are taken into account). The value of n influences precision of the abstraction. Of course, when a pool of threads (a set of threads started once and used for several tasks) is used, the computation of the hash value must be restarted immediately after picking the thread up from the pool.

3.5 A Comparison of Coverage Metrics

In this section, a comparison of our newly proposed coverage metrics with selected existing metrics is presented. Our architecture for collecting concurrency-related coverage is built upon the IBM Java Concurrency Testing Tool (ConTest) introduced in Chapter 2. To recall, the tool provides a facility for byte-code instrumentation and a listeners infrastructure allowing one to create *plug-ins* for collecting various pieces of information about the multi-threaded Java programs being executed as well as to easily implement various algorithms for dynamic analyses. The tool is itself able to

[1]Instances of variables are identified by an object identifier and the appropriate field of the object.

Table 3.2: Test cases

	Classes	kLOC	Error Manifestation Ratio	Error type
Dining phil.	2	0.1	0.4151	deadlock
Airlines	8	0.3	0.0333	atomicity viol.
Crawler	19	1.2	0.0006	atomicity viol.
Rover	83	5.4	0.0005	deadlock
FtpServer	120	12.2	0.4032	data race
TIDOrbJ echo	1 399	84.3	0.0170	none / data race

collect structural coverage metrics (basic blocks, methods) and some concurrency-related metrics (ConcurPairs, Sync) too. ConTest further provides a noise injection facility which allows one to observe different legal interleavings if the test is executed repeatedly. We use our platform called SearchBestie described in Appendix B to set up and execute tests with ConTest, and to collect, maintain, and export results produced by ConTest and its plug-ins from multiple executions of a test.

Test Cases

We have evaluated the metrics discussed in the previous section on five small test cases (Dining philosophers, Airlines, Crawler, Rover, FtpServer) and one bigger test case (TIDOrbJ). Table 3.2 shows for each test case its size expressed by the number of instrumented classes and number of lines of code (LOC). The next column shows the probability of spotting an error during a test execution when random noise injection is used (computed as the number of executions where an error occurs divided by the total number of executions). The last column shows the type of error present in the test case.

The *Dining philosophers* test case is an implementation of the well-known synchronisation problem of dining philosophers. Our implementation is taken from the distribution of the Java PathFinder model checker. The program generates a set of 6 philosophers (each represented by a thread) and the same number of shared objects representing forks. A deadlock can occur when executing the test case.

The *Airlines* test case is a simple artificial program simulating an air ticket reservation system. It generates a database of air tickets and then allows 2 resellers (each represented by a separate thread) to sell tickets to 4 sets of 10 customers (each set is represented by a separate thread). Finally, a check whether the number of customers with tickets is equal to the number of sold tickets is done. The program contains a high-level atomicity violation whose occurrence makes the final check fail.

The four other considered programs are real-life case studies. *Crawler* is a part of an older version of a major IBM production software. It demonstrates a tricky concurrency bug detected in this software. The crawler creates a set of threads waiting for a connection. If a connection simulated by a testing environment is established, a worker thread serves it. There is an error in a method that is called when the crawler shuts down. The error causes an exception sometimes leading to a deadlock. The trickiness of the error can be seen from its very low error probability shown in Table 3.2.

The second real-life case study is a Java version of the NASA Ames K9 Rover Executive whose verification is presented in [42]. The Rover Executive is a platform for controlling autonomous vehicles called rovers. The test case executes three high-level plans—programs written in a language that specifies actions and constraints on the movement, experimental apparatus, and other resources of the rover. The test case contains a deadlock causing the execution of the test to hang. This error is also very difficult to spot as can be seen from very low error manifestation probability. Moreover, the test case also contains rarely manifested high level atomicity violation in the method responsible for changing the current plan of the rover. The old plan is set to `null` while some other thread still access it. Such situation leads to the `NullPointerException`. This error does not manifested during any performed testing run on systems with two cores used in this chapter. We detect the error only when executing the test case with specific noise setting (for more information see Chapter 4).

Our third real-life case study is an early development version of an open-source *FtpServer* produced by Apache. This case study has 120 classes. The server creates a new worker thread for each new incoming connection to serve it. The version of the server we used contains several data races that can cause exceptions during the shut down process when there is still an active connection. The probability of spotting an error when noise injection is enabled is quite high in this example because there are multiple places in the test where an exception can be thrown.

Our biggest test case is *TIDOrbJ* which is a CORBA-compliant ORB (Object Request Broker) software that is a part of the MORFEO Community Middleware Platform [99]. The instrumented part of the middleware has 1399 classes. We have used the *Echo concurrent* test which checks how the infrastructure handles multiple concurrent simple requests. The test starts an instrumented server and then 10 clients, each sending 5 requests to the server. There was originally no error in this test, and therefore we introduced one by commenting one `synchronised` statement in the part of code that is executed by the test. This way, we introduced a high-level atomicity violation that leads to a `NullPointerException`.

Experimental Setup

We used our infrastructure introduced above to collect relevant data from 10,000 executions of the small test cases and 4,000 executions of TIDOrbJ. In order to see as many different legal interleaving scenarios as possible, we set up ConTest to ran-

Table 3.3: Test cases and abstract identifiers

	ObjectAbstraction			ThreadAbstraction			
	Real	Type	Abs	Real	Type	Abs$_{10}$	Abs$_{20}$
Dining phil.	130	3	3	7	2	2	2
Airlines	15 210	6	6	60	3	3	4
Crawler	1 828	13	14	180	4	9	12
Rover	6 150	48	53	80	8	10	16
FtpServer	26 110	27	29	1 641	5	5	6
TIDOrbJ echo	180 320	98	129	79	5	9	11

domly inject noise into the executions. We have implemented ConTest plug-ins to collect coverage information and set up SearchBestie to detect occurrences of errors (deadlocks were detected using a timeout, other errors by detection of unhandled exceptions). All further studies of the metrics were done using the collection of executions obtained this way. For instance, we often needed to evaluate the behaviour of the metrics on series of executions. To generate the needed series of executions, we used SearchBestie to randomly select a needed number of executions out of the recorded collection and to compute accumulated values of the chosen metric on such series. All tests were executed on a computer with an Intel 6600 processor and 2 GB of memory, running Sun Java version 1.6 under GNU Linux.

Object and Thread Abstract Identification

Table 3.3 summarises information on our test cases (both from the point of view of the source code as well as the runtime behaviour) and—most importantly—it illustrates the effect of our abstract object and thread identifiers. The table provides information about the size of the case studies in terms of the numbers of threads and objects created in them. The table also illustrate precision of our abstract identifiers of objects and threads. The *Real* column contains the total number of distinct objects (or threads) we encountered in 10 performed executions of the tests. The *Type* column shows the total number of distinct object (or thread) types we have spot, and the *Abs* columns show the total number of distinct abstract objects (or threads) we distinguish using our abstract identifiers introduced in Section 3.4. For the thread abstraction, two values are given showing the influence of the length n of the considered sequence of methods called by the threads.

Typical Saturation Behaviour of the Metrics

To decide whether a coverage metric is suitable for saturation-based testing or not, one needs to evaluate several aspects of its behaviour. The typical behaviour of the

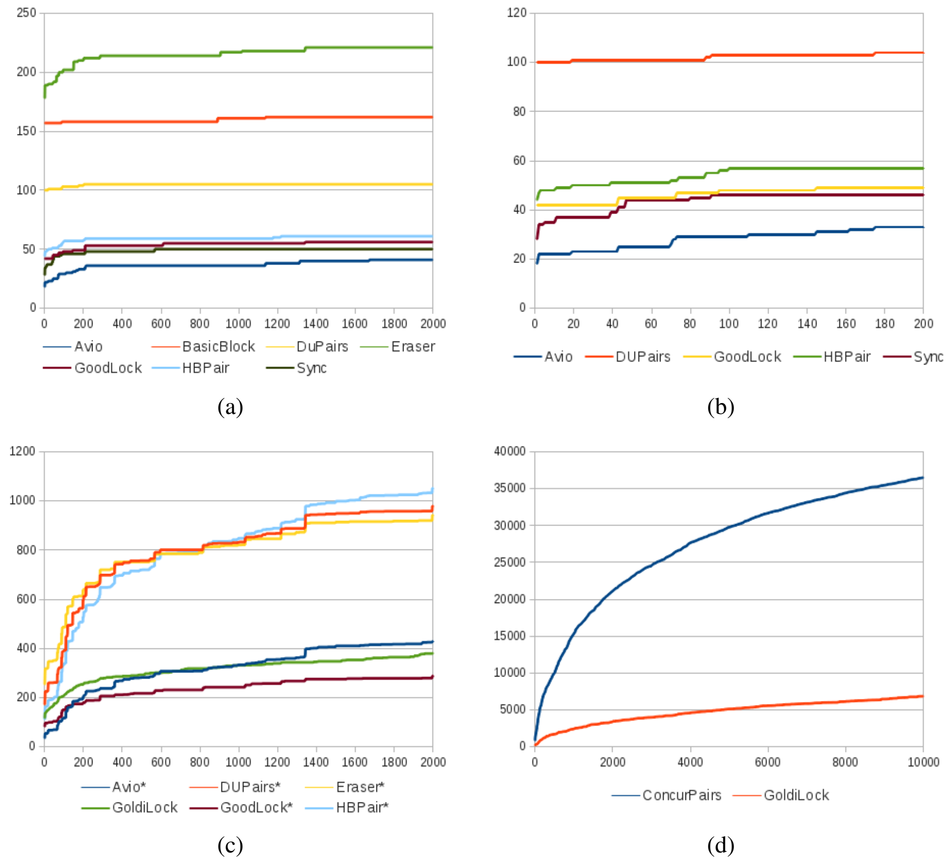


Figure 3.1: Saturation of different metrics on the Crawler test case (the horizontal axis gives the number of executions, the vertical axis gives the cumulative number of covered tasks)

considered coverage metrics can be seen in Figure 3.1. All four sub-figures show the cumulative number of coverage tasks of the metrics covered during one randomly chosen series of the Crawler test case executions (with the thread abstraction variable *n* set to 20).

Figure 3.1(a) shows the behaviour of the metrics that, according to our opinion, do not capture the concurrent behaviour accurately enough. One coverage metrics for non-concurrent code measuring the number of *basic blocks* covered during tests is added to demonstrate the difference between classical and concurrency-related coverage metrics. The coverage obtained under the metric based on basic blocks is nearly constant all the time because we are repeatedly executing the same code with the same inputs. For the rest of the metrics shown in Figure 3.1(a), the cumulative number of tasks covered during test executions increases only within approximately the

200 first executions, and then a saturation is reached. The only metric which slightly differ from the others in this group are Eraser and DUPairs. The Eraser metric has a similar behaviour to the *Avio* metric (and the metrics close to it) but approximately four times higher numbers of covered tasks. This is caused by the fact that the tracked shared variables usually get to four Eraser states. The DUPairs metric has also higher numbers of covered tasks but it is almost all the time stabilised.

The most interesting part of Figure 3.1(a) which is between 0 and 200 executions is zoomed in Figure 3.1(b). One can see that the saturation effect occurs earlier (at about 100 executions) for the HBPair and Sync metric which both focus on synchronisation events only. The Avio metric (and also the Eraser metric which is not shown) that focus on accesses to shared variables saturate a bit later. The depicted curves demonstrate one further disadvantage of the concerned metrics—a presence of distinctive shoulders. A repeated execution of the test case does examine different concurrent behaviours (which is indicated by the later discussed metrics) but the metrics concerned in the figure are not able to distinguish differences in these behaviours, and therefore we can see clear shoulders in the curves (i.e., sequences of constant values). The presence of such shoulders makes automatic saturation detection harder.

Figure 3.1(c) demonstrates a positive effect of considering an extended context of the tracked events as proposed in Section 3.3. The metrics concerned in this subfigure (i.e., Avio*, Eraser*, DUPairs*, HBPair*, GoodLock*, and GoldiLock) are able to distinguish differences in the behaviour of the executed tests more accurately, leading to shorter shoulders, bigger differences in the cumulated values, and a later occurrence of the saturation effect—indicating that the concerned metrics behave in a way much better for saturation-based testing. As can be seen from a similar jump in the obtained coverage of the HBPair*, Eraser*, and Avio* metrics at around 1300 executions, the extended context can sometimes have a dramatic influence. The jump is caused by the abstract thread identifiers. At the given point, a thread with a new abstract identifier appears, and all tasks involving this thread are different to those already known. This leads to a much more significant increase in the cumulative coverage. A special attention should be paid to the GoldiLock metric. This metric does not suffer from shoulders nor sudden, dramatic increases of the obtained coverage, and it reaches saturation near the saturation points of the other metrics. This is a very positive behaviour, and the GoldiLock metric is clearly winning here.

Figure 3.1(d) shows problems of metrics that are too accurate, namely, ConcurPairs and GoldiLock*. These metrics work fine for small test cases but when used on a bigger test case they tend to saturate late and produce enormous numbers of covered tasks.

Quantitative Properties of the Metrics

Quantitative properties of the considered metrics in all our test cases can be seen in Table 3.4. In particular, Table 3.4 shows, for each metric and each test case, three

Table 3.4: A quantitative comparison of the metrics

	Dining phil.			Airlines			Crawler		
	Total	Average %	Smooth %	Total	Average %	Smooth %	Total	Average %	Smooth %
Avio	6	47	0	17	60	1	40	22	1
Avio*	30	10	0	490	2	10	418	3	9
ConcurP.	4 059	6	38	16 730	6	85	20 866	3	83
DUPairs	18	76	0	43	97	0	105	81	1
DUPair*	72	19	0	1 401	3	9	921	11	8
Eraser	29	76	0	73	96	0	217	64	2
Eraser*	89	25	0	1 429	5	8	861	19	5
GoldiLock	26	73	0	102	64	2	384	20	12
GoldiLock*	119	16	0	4 217	1	20	3 335	3	26
GoodLock	9	56	0	0	-	0	57	52	1
GoodLock*	22	23	0	0	-	0	258	17	4
HBPair	6	62	0	25	79	0	61	39	1
HBPair*	29	13	0	1 013	2	13	984	4	12
Sync	8	56	0	27	78	0	49	46	1

	Rover			FtpServer			TIDOrbJ echo		
	Total	Average %	Smooth %	Total	Average %	Smooth %	Total	Average %	Smooth %
Avio	178	71	1	529	45	10	822	50	8
Avio*	1 472	10	8	1 023	33	16	3 280	29	22
ConcurP.	120 623	2	100	526 280	6	100	4 908 100	2	100
DUPairs	478	97	0	330	92	2	1 933	98	2
DUPair*	2460	21	4	646	82	3	3 092	90	4
Eraser	840	94	1	684	88	4	2 978	90	4
Eraser*	3 352	29	4	1 086	79	4	4 886	83	6
GoldiLock	638	89	1	1 091	61	9	6 265	51	29
GoldiLock*	14 898	4	13	2 210	47	12	10 434	41	46
GoodLock	45	58	0	0	-	0	321	63	3
GoodLock*	137	24	2	0	-	0	915	34	6
HBPair	70	79	1	13	73	0	131	70	2
HBPair*	721	8	5	28	49	0	420	46	5
Sync	69	72	1	22	66	0	172	79	2

values computed from a set of 100 different random series consisting of 2,000 test executions. The columns labelled as *Total* show the average total number of distinct tasks produced by the metric. This number demonstrates a big disadvantage of the ConcurPairs coverage metric, namely, its problem with scalability. The metric produced nearly 5 million of distinct tasks for 2,000 executions of the TIDOrbJ test case which makes further analyses quite time demanding.

The columns of Table 3.4 labelled as *Average percentage* represent the ratio between the average number of tasks covered within one execution and Total. A high number in this column means that most of the total number of covered tasks were covered within one execution. The cumulative coverage under such metrics (for DUPairs, Eraser, and Sync) usually stabilises early or grows very slowly. In both of these cases, the detection of saturation is difficult. Contrary, if the average percentage is too low (for ConcurPairs and GoldiLock*), the cumulative coverage grows for a very long time.

Finally, the columns of Table 3.4 labelled *Smooth percentage* give an insight in how smooth the growth of the accumulated coverage is. The column contains the ratio between the average number of the distinct cumulative coverage values reached under a metric when going through the considered executions and the number of test executions (2,000). High values (for ConcurPairs and GoldiLock*) mean that the cumulated coverage under the metric changed many times, and therefore there was contiguously growing. Low values (for Avio, DUPairs, Eraser, GoodLock, and Sync) mean that the cumulated coverage changed only a few times, and therefore there either occurred a fast saturation or there appeared long shoulders. Both of these phenomena are problematic for a good metric to be used in saturation-based testing.

The table also shows a disadvantage of the GoodLock* metric. The metric focuses on nested locking as was described in Section 3.3. If such a phenomenon does not occur in the tested program, the metric provides no information as can be seen in the Airlines and FtpServer test cases. On the other hand, the metric can provide additional information which cannot be directly inferred by other metrics in programs which contains this phenomenon. In total, the evaluation in Table 3.4 gives similar champions for a good metrics to be used in saturation-based testing as what we saw in Figure 3.1(c). Namely, this is the case of the Avio*, Eraser*, DUPairs*, HBPair*, and GoldiLock metrics.

3.6 Summary

In this chapter, a methodology of deriving new coverage metrics to be used in testing of concurrent software from dynamic (and possibly also static) analyses designed for discovering bugs in concurrent programs has been proposed. Using this idea, several new particular metrics have been derived. And, an empirical evaluation of these metrics have been performed, which has shown that several of them are indeed better for use in saturation-based and search-based testing than various previously known metrics.

As an additional advantage of the metrics that we have proposed, their better applicability in debugging cab be mentioned. For debugging, understandability of each coverage task is important. We believe that the tasks generated by our metrics provide more concurrency problem-related information to the tester than existing metrics such as ConcurPairs or DUPairs. The tester can track the threads and objects that appear in the covered tasks to their place of creation or use some additional information (e.g., a lockset) present in the tasks to better understand what happened during the witnessed executions.

Chapter 4

Noise Injection Heuristics

In this chapter, a new heuristics for noise injection which uses coverage information to select places in an execution of a given code where to put a noise is proposed. Furthermore, a way to determine the strength of the noise needed to suitably affect the behaviour of tested programs is also proposed. Next, in this chapter, the current lack of experimental evaluation of the various existing noise seeding and noise placement heuristics is addressed. Namely, the chapter provides a *systematic comparison* of several noise injection techniques available in the IBM Concurrency Testing Tool (ConTest), which represents the state of the art of noise injection, as well as the newly proposed heuristics on a set of test cases of different size. The comparison is based on the coverage obtained under two selected concurrency coverage metrics, the needed execution time, and the rate of manifestation of concurrency errors in the testing runs. The comparison shows that there is no silver bullet among the many existing noise seeding and noise placement heuristics. Moreover, it identifies weak and strong aspects of the different heuristics in different contexts and can thus serve as a guide for a user which intend to apply the heuristics in the testing process. Apart form that, the comparison also shows that the newly proposed heuristics may in certain cases provide an improvement in the testing process.

The rest of the chapter is organised as follows. Existing noise injection heuristics are discussed in Section 4.1. Section 4.2 introduces a new noise placement heuristics. And, Section 4.3 provides a comparison of different noise seeding heuristics available in IBM ConTest and a comparison of various noise placement heuristics including the newly proposed heuristics. The maximal improvement achieved by the noise-based testing approach in our test cases are presented at the end of this section. Finally, Section 4.4 provides a summary the obtained results and based on them gives suggestions on how to test programs using the noise injection technique.

4.1 Existing Noise Seeding and Placement Heuristics

Existing works discuss three main aspects of heuristic noise injection: (i) How to make noise, i.e., which type of noise generating mechanism should be used, (ii) where to inject noise during a test execution, i.e., at which program location and at which of its executions (if it is executed multiple times), and (iii) how to minimise the amount of noise needed for manifestation of an already detected error when debugging. This chapter mainly targets the first two aspects. The debugging problem is discussed in Chapter 5.

There exist several ways how a scheduler decision can be affected in Java as was already described in Chapter 2. Recall that the noise maker can use calls of `yield()` to cause a context switch or `sleep()` and `wait()` to cause a delay. The current version of the IBM ConTest tool [31] comes with several more noise seeding techniques also mentioned in Chapter 2. The *synchYield* technique combines the yield technique with entering a monitor that is shared among all threads, the *busyWait* technique loops for some time, the *haltOneThread* technique occasionally stops one thread until any other thread cannot run, and finally, the *timeoutTampering* heuristics randomly reduces the time-out used when calling `sleep()` in the tested program.

There also exist multiple noise placement techniques for determining where to put a noise. The problem of noise placement can be divided into two subproblems: (i) Which program locations are suitable for noise injection and (ii) which particular occurrence of selected program locations in an execution of the program actually affect by the noise.

IBM ConTest allows to inject a noise before and after any concurrency-related event (including, accesses to class member variables, static variables, and arrays, and the calls of `wait`, `interrupt`, `notify`, `monitorenter`, and `monitorexit` routines). The *rstest* tool [101] considers as possibly interesting places before concurrency-related events only. Moreover, *rstest* uses a simple escape analysis and a lockset-based algorithm to statically identify the so-called *unprotected accesses* to shared variables. An unprotected access reads or writes a variable which is visible to multiple threads without holding an appropriate lock. This optimisation reduces the number of places where the noise can be put but suppresses ability to detect some concurrency errors, e.g., high-level data races or deadlocks where all accesses to problematic variables are correctly guarded by a lock.

It is discussed in [31, 35, 101] that putting noise on every possible place is inefficient. These works also claim that only a few relevant context switches are critical for the concurrency error. Moreover, putting noise to a certain place in the execution can either help to spot the concurrency error or mask it completely as we presented in our previous works, e.g., [58]. Therefore, several heuristics for choosing places where to put a noise were proposed, e.g., in [31, 101, 16, 36, 107].

The simplest heuristics which targets the problem of selecting a particular occurrence of the considered program locations is based on *random noise* [31, 101]. This heuristics puts a noise before/after an executed program location *ploc* with a given probability. The probability is the same for all *ploc*s in the execution. It was shown in [16] that focusing random noise only on a single variable over which a data race exists increases the probability of spotting the error. The authors also propose a heuristics which helps to choose a suitable variable without additional information from a data race detector. In [36], several *concurrency anti-patterns* are discussed, and for each of them, a suitable scheduling scenario that leads to a manifestation of the corresponding concurrency error is presented, but the paper contains no practical evaluation of the proposed heuristics.

Only a few heuristics based on concurrency coverage models have been published. Coverage-directed generation of interleavings presented in [31] considers two coverage models. The first model determines whether the execution of each method was interrupted by a context switch. The second, a bit refined model determines whether a method execution was interrupted by another instance of the same method simultaneously executed by a different thread or by any other method simultaneously executed by a different thread. This model is motivated by the observation that sometimes a concurrency-related error manifests if two threads are simultaneously executing the same part of code containing the error.

In [107], a heuristics for increasing the *synchronisation coverage* introduced in Chapter 3 is sketched. The synchronisation coverage metrics measure whether for each program location *ploc* where a monitor is entered the thread was *blocked* and thus cannot proceed, whether the thread after obtaining the monitor blocked some other thread trying to obtain the same monitor (therefore act as *blocking* thread), or no interference among threads occur and therefore the *ploc* was just *visited*. The heuristics makes a thread sleep for a while before and/or after executing *ploc*. The authors claim that this can increase the synchronisation coverage but no experimental evaluation is provided in the paper. None of these two coverage-based heuristics focuses on accesses to shared variables which can limit their ability to discover some concurrency errors, e.g., data races.

Finally, note that modern deterministic testing tools like CHESS introduced in Chapter 2 block all threads except one which is enabled for running. This could also be considered as an invasive noise technique. The algorithms used in these techniques choose threads to enable for running with intention to maximise coverage of a considered model of the tested program. Therefore, in a sense, the techniques implement coverage-based noise heuristics too.

4.2 A New Coverage-based Noise Placement Heuristics

We are now going to describe our proposal of a new coverage-based heuristics for selection of places where to inject noise during a test run takes into account that the noise can be caused by any of the noise seeding techniques and it considers only program locations *ploc*s that appear before concurrency-related events (considered by IBM ConTest) as suitable for noise injection. The heuristics targets both accesses to shared variables as well as the use of synchronisation primitives. The goal is to be able to discover all kinds of concurrency errors. The heuristics monitors the frequency of a *ploc* execution during a test and puts a noise at the given *ploc* with a probability biased wrt. this frequency—the more often a *ploc* is executed the lower probability is used. Furthermore, the heuristics also derives the strength of a noise to be used from the timing of events observed in previous executions of the test (although for determining the strength of noise, alternative approaches can be used too).

The main principle of the heuristics is sketched first and then we explain it in more details. The heuristics is incorporated into the testing process as follows: (i) A test is run with no noise, and a set of covered tasks of the coverage metric described below together with information on relative timing of appearance of monitored concurrency-related events are generated during the first execution of the test. (ii) A set of the so-called *noise tuples* defined later is generated from the gathered information. (iii) Random noise at the *ploc*s included in the noise tuples is generated, and the average frequency of execution of these *ploc*s within particular threads is gathered during the next test execution. (iv) Biased random noise of strength computed wrt. the collected statistics is (repeatedly) produced at the collected *ploc*s. Coverage information is updated during each execution, and new noise tuples are constantly learnt. Likewise, all other collected statistics are updated during each test run. Due to performance reasons, only one thread is influenced by noise at a time.

The coverage metric used by the heuristics considers coverage tasks of the form $(t_1, ploc_1, t_2, ploc_2)$. There are two situations when such task is covered. First, a task is covered if a thread t_1 accesses a shared variable v at $ploc_1$, and subsequently a thread t_2 accesses v at $ploc_2$, which is a typical scenario critical for occurrence of concurrency-related errors. If t_1 owns a monitor when accessing v at $ploc_1$, another task $(t_1, ploc_3, t_2, ploc_2)$ where $ploc_3$ refers to the location where t_1 obtained the last monitor is also covered. This is motivated by considering the relative position of locking a critical section in one thread and using it in another thread as important. Second, a new task is covered if a thread t_1 releases a monitor obtained at $ploc_1$, and subsequently a thread t_2 obtains the monitor at $ploc_2$. Each covered task is annotated by the number of milliseconds that elapsed between the events on which the task is based. The threads are identified in an abstract way based on the history of their creation in the same way as already described in Chapter 3.

The proposed coverage metric differs from metrics introduced in Chapter 3. The metric does not consider internal state of any detection algorithm. The metric focuses on occurrence of events important for most of detection algorithms—that is locking and accesses to shared memory. But, the stress is put on covering situations where insertion of a noise could improve the testing process by observing different interleavings.

Our heuristics injects noise before a location $ploc_1$ executed by a thread t_1 if a task $(t_1, ploc_1, t_2, ploc_2)$ has been covered within some previous execution. This way, our heuristics tries to reverse the order in which the instructions at the considered program locations are executed. The coverage information collected during previous runs is transformed into *noise tuples* of the form $(t_1, ploc_1, min, max, orig, exec)$. Here, t_1 identifies a thread and $ploc_1$ the program location where to put a noise. The next two values give the minimal and maximal number of milliseconds that elapsed between the events defining the given coverage task. The length of a delay is randomly chosen within the interval (min, max). These values can be used for determining the strength of noise to be used as a delay of length randomly chosen from between the values. If there are multiple coverage tasks with the same couple $(t_1, ploc_1)$, i.e., with different couples $t_2, ploc_2$, *min* and *max* are computed from all such tasks. The *orig* value contains an identification of the run where the couple $(t_1, ploc_1)$ was spot for the first time. In order to limit values of *min* and *max*, their update is possible only within a limited number of test executions after the *orig* run. Finally, the *exec* value contains the average number of times the couple $(t_1, ploc_1)$ is executed during a test execution. It is used to bias the probability of noise injection at $ploc_1$.

In repeated executions of a test, the so far computed noise tuples are loaded, and the noise is generated at program locations given by them with the probability computed from the number of times the locations have been executed (the *exec* value). The base probability is computed using the following expression:

$$prob = max(0.001, \frac{1}{4 * exec})$$

The base probability is therefore obtained as maximal value from the minimal noise probability accepted by ConTest which is 0.001 and an expression which sets the probability for often executed program locations to a lower values. The probability is in such case further divided by 4 to keep the noise injection frequency reasonably low (25 % for a *ploc* which is executed once during each test). This is motivated by our observation that a higher probability than 25 % degrades the test performance and usually does not provide considerably better results as can be seen from the comparison presented in Section 4.3. If the *exec* value is not yet available, the probability of 0.01 is used.

4.3 A Comparison of Noise Injection Techniques

This section presents an experimental comparison of selected noise seeding and noise placement heuristics available in ConTest as well as of the above newly proposed heuristics. The section is divided into four sub-parts: (i) First, a description of the testing environment and the test cases used for the comparison is given. (ii) Then, the comparison of different noise seeding heuristics is provided. The comparison first compares the basic noise seeding heuristics according to their efficiency to improve detection of concurrency errors, to improve concurrency-related coverage metrics in the considered test cases, and to affect test execution time. Then, the improvement which can be achieved by a combination of basic noise seeding techniques with advanced seeding techniques *haltOneThread* and *timeoutTampering* is studied. (iii) Next, a comparison of different noise placement heuristics is provided. The comparison first compares the noise placement heuristics according to ability to detect concurrency errors and provide high concurrency-related coverage. Then, a comparison of the heuristics using relative results where the total number of covered tasks or detected errors is divided by the time the heuristics needed to achieve the results is provided. In effect, therefore, this comparison of noise placement heuristics punishes techniques that either put too much noise into test executions or provide poor results only. (iv) Finally, the best improvement achieved by the noise-based testing in the considered test cases is presented.

4.3.1 Testing Environment Used for Comparison

The comparison was done using 5 test cases. The Airlines, Crawler, FTPServer, and TIDOrbJ test cases were already described in Chapter 3. This time, the TIDOrbJ test case was not modified to contain any concurrency error. The *Sunbank* test case was derived from a simple example of a concurrent program used in a tutorial published by Sun Microsystems. The test case simulates bank accounts where multiple accounting threads perform simple changes to the particular accounts and the total balance of the bank without a proper synchronisation over the global balance. The test case consists of 3 classes, has 0.1 kLOC, and contains a data race.

During each test run, coverage wrt. two chosen metrics were measured—namely, *Avio** which measures the number of different witnessed interleavings among subsequent accesses to shared variables and *HBPair** which measures the number of successful synchronisations among program threads that establish a happens-before relation among events in these threads. Both concurrency-related coverage metrics were described in more details in Chapter 3. These metrics have been chosen due to their very good ratio of providing good results from the point of view of suitability for saturation-based or search-based testing and a low overhead of measuring the achieved coverage (and hence their suitability for performing many tests with a minimal interference with the tested programs). Note that both of the metrics that are used for evaluation of the testing are different than the specialised metrics that has been proposed above as a means for driving the noise injection.

Besides the coverage information, execution times and occurrences of un-handled exceptions which were thrown when a known concurrency error occurred were monitored. Collection of this information of course affects thread scheduling of the monitored test cases, but the influence is the same for all performed executions. The

Sunbank, Airlines, Crawler, and FTPServer test cases were run on Intel Core2 6600 machines. The TIDOrbJ test case was run on a machine with two Intel X5355 processors. All machines run 64-bit Linux and Java version 1.6.

The SearchBestie infrastructure described in Appendix B which is able to collect results produced by IBM ConTest was used to collect data for the comparison. To recall from Chapter 2, IBM ConTest provides five basic techniques for noise seeding: *yield*, *sleep*, *wait*, *busyWait*, and *synchYield*. In addition, the so-called *mixed* technique simply randomly chooses one technique from the others. The mentioned basic noise seeding techniques can be combined in ConTest with two advanced noise seeding techniques—*haltOneThread* and *timeoutTampering*.

The default noise placement heuristics used by IBM ConTest is *random* which selects places where to generate noise randomly. Moreover, ConTest also provides 4 different versions of heuristics which restrict the noise generation to events related to (certain) shared variables (the *sharedVar* heuristics). The probability of causing a noise at a selected *ploc* is driven by the *noise frequency* (*nFreq*) parameter ranging from 0 (no noise) to 1000 (always). We limit this parameter to values 0, 50, 100, 150, and 200. Higher values cause in some cases a significant performance degradation and are therefore not considered. Finally, two versions of the newly proposed heuristics were considered. The basic version of the heuristics selects places according to collected coverage information and then uses the *nFreq* parameter to control the probability of causing a noise. The second version additionally uses the inferred *min* and *max* values to compute the strength of noise.

Data for the comparison were collected as follows: Each of our 5 test cases was tested with 496 different noise injection configurations derived from the possible combinations of parameters of the heuristics described above. Data from 60 executions for each configuration were collected. Then, SearchBestie was used to produce for each of the 496 configurations 100 randomised sequences. Finally, from each set of sequences, SearchBestie computed average accumulated values for sequences of 10, 20, 30, 40, and 50 results (the length of the sequence is denoted as *SeqLen* below). These average accumulated results used for the comparison therefore represent average values that one obtains when executing the given configuration *SeqLen* times.

The comparison of noise seeding techniques compares the heuristics according to their ability to improve the testing process. Therefore, each test case was executed 100 times without any noise injection heuristics enabled but with the collection of data about the testing enabled, which also influences the scheduling. Then, average values for particular sequence lengths computed from 100 randomised sequences of the obtained results were computed. The values were used as a bottom-line values for the comparison of noise seeding techniques.

Table 4.1: Average relative improvement of error detection when using different types of noise (timeoutTamper and haltOneThread disabled, seqLen 50)

Test	nFreq	Noise Type						aver.
		yield	sYield	sleep	wait	bWait	mixed	
Sunbank	50	1.88	0.87	3.23	**4.43**	2.60	1.70	2.45
	100	0.82	0.00	5.90	**8.27**	4.87	6.97	4.47
	150	5.28	0.00	**8.23**	4.97	3.27	4.03	4.30
	200	4.66	0.00	7.60	**12.07**	6.13	4.93	5.90
Airlines	50	**3.02**	0.34	1.27	1.30	2.25	0.82	1.50
	100	**3.33**	1.77	1.88	1.68	0.89	2.70	2.04
	150	**2.81**	2.66	0.41	0.84	1.79	0.95	1.58
	200	**3.19**	1.86	1.80	1.16	0.46	1.30	1.63
FTPServer	50	**0.94**	0.93	0.35	0.58	0.34	0.48	0.60
	100	**1.02**	0.92	0.24	0.32	0.49	0.37	0.56
	150	**1.02**	**1.02**	0.34	0.31	0.27	0.30	0.54
	200	0.92	**0.99**	0.25	0.31	0.61	0.34	0.57

4.3.2 A Comparison of Noise Seeding Heuristics

This comparison studies the influence of the different noise seeding techniques and the noise frequency on how the testing results are improved in comparison to testing without noise injection. Since ConTest does not allow one to use its advanced noise seeding techniques *timeoutTampering* and *haltOneThread* without one of its basic noise seeding techniques, the effect of the basic noise seeding techniques which are activated via the `noiseType` parameter of ConTest is studied first. Then, the effect of the *timeoutTampering* and *haltOneThread* seeding techniques is focused. This comparison shows which combinations of noise seeding heuristics and their parameters provide the best improvement.

A Comparison of the Basic Noise Seeding Techniques

Our comparison of basic noise seeding techniques is presented in Tables 4.1, 4.2, 4.3, and 4.4. The tables study the basic noise seeding techniques available in IBM ConTest and the influence of the *nFreq* parameter. The tables summarise results obtained when both the *haltOneThread* and *timeoutTampering* seeding techniques were disabled and the random noise placement heuristics was enabled. Each cell contains an average value computed from 100 sequences of the considered length (50) with a particular configuration. The comparison is done using the error manifestation ratio, Avio* coverage, HBPair* coverage, and execution time. The results for *seqLen*=50

Table 4.2: Average relative improvement of the Avio* coverage when using different types of noise (timeoutTamper and haltOneThread disabled, seqLen=50)

		Noise Type						
Test	nFreq	yield	sYield	sleep	wait	bWait	mixed	aver.
Sunbank	50	**1.14**	1.06	0.79	1.01	0.82	0.81	0.94
	100	**1.03**	0.91	1.02	0.90	0.92	1.00	0.96
	150	**1.07**	0.90	1.01	1.00	1.00	1.00	1.00
	200	**1.12**	1.01	1.01	1.02	1.00	1.02	1.03
Airlines	50	**1.25**	0.90	0.51	0.75	0.84	0.75	0.83
	100	**1.14**	0.51	0.51	0.51	0.51	0.51	0.61
	150	**1.41**	0.95	0.51	0.51	0.50	0.84	0.79
	200	**1.04**	0.96	0.51	0.48	0.51	0.51	0.67
Crawler	50	1.00	1.00	1.14	1.36	**1.40**	1.14	1.17
	100	1.00	1.00	1.38	1.43	**2.05**	1.37	1.37
	150	1.00	1.00	1.41	1.43	**1.71**	1.35	1.32
	200	1.00	1.00	**1.64**	1.43	1.53	1.29	1.32
FTPServer	50	1.01	1.06	1.12	1.11	1.10	**1.14**	1.09
	100	1.04	1.00	1.19	1.14	1.19	**1.20**	1.13
	150	1.02	1.03	1.14	1.07	1.04	**1.16**	1.08
	200	1.04	1.03	1.05	1.06	1.04	**1.18**	1.07
TIDOrbJ	50	0.94	**1.00**	0.92	0.91	0.95	0.95	0.94
	100	**0.99**	0.98	0.90	0.91	0.97	0.94	0.95
	150	0.98	**1.02**	0.89	0.89	0.85	0.93	0.93
	200	0.99	**1.02**	0.87	0.86	0.82	0.93	0.91

are presented. The results for other values of this parameter provide similar tendencies but the difference among values is highest for the *seqLen* set to 50. Therefore, the results for this parameter are presented. The best improvement (or lowest degradation respectively) for each test case and each *nFreq* parameter is highlighted in bold.

The improvement presented in the following tables may seem to be low in some cases. This is caused by use of the random noise placement heuristics which is in some cases inefficient as is shown in the comparison of noise placement heuristics below. We choose this heuristics for the comparison of noise seeding techniques because it should not favour any of the considered heuristics. The potential benefit obtained by the noise injection technique can be seen later in Table 4.18 which presents maximal average improvement for each test case and each metric together with a configuration of noise injection heuristics used to achieve the improvement.

Table 4.3: Average relative improvement of the HBPair* coverage when using different types of noise (timeoutTamper and haltOneThread disabled, seqLen=50)

		Noise Type						
Test	nFreq	yield	sYield	sleep	wait	bWait	mixed	aver.
Airlines	50	**1.20**	0.89	0.51	0.74	0.81	0.74	0.82
	100	**1.15**	0.52	0.47	0.49	0.51	0.51	0.61
	150	**1.40**	0.97	0.52	0.48	0.47	0.82	0.77
	200	**1.03**	0.95	0.49	0.52	0.46	0.48	0.65
Crawler	50	0.97	0.88	1.09	1.11	**1.26**	0.96	1.05
	100	0.99	1.13	1.17	1.17	**1.49**	1.15	1.18
	150	0.89	1.08	**1.17**	1.14	**1.17**	1.16	1.10
	200	1.00	1.04	1.16	1.15	1.12	**1.18**	1.11
FTPServer	50	0.94	1.16	1.16	1.10	1.17	**1.23**	1.13
	100	1.08	1.14	1.17	1.21	**1.44**	1.25	1.22
	150	1.12	1.02	1.21	1.10	1.21	**1.24**	1.15
	200	1.08	1.15	1.16	**1.25**	1.16	**1.25**	1.18
TIDOrbJ	50	0.92	0.97	0.96	**0.98**	**0.98**	**0.98**	0.96
	100	0.97	0.98	0.95	0.97	**1.00**	0.97	0.97
	150	0.95	0.93	0.90	0.90	0.92	**0.98**	0.93
	200	0.98	**1.03**	0.90	0.90	0.95	0.96	0.95

Table 4.1 shows the average relative improvement of error detection that we observed when using different basic noise seeding techniques available in ConTest[1]. Additionally, we also consider the ConTest setting which randomly chooses among basic noise seeding techniques before each program location (referred as *mixed* in the table). The entries of the table therefore give the ratio of the number of error manifestations observed when using noise injection of the respective type against the number of error manifestations without any noise setting enabled. Moreover, average values (denoted as *aver.* in the table) are provided for a better comparison. Values lower than 1.00 mean that the appropriate configuration provided a worse result than without noise. Higher values mean that noise of the appropriate type provides better results. For instance, 1.25 means that the given type of noise on average detected an error by 25 % more often. Results for the Crawler and TIDOrbJ test cases are omitted because the error in the Crawler test case was not detected by random noise placement heuristics and there is no error in the TIDOrbJ test case.

[1]The numbers in Table 4.1 slightly differ from numbers presented in paper [63] because we discovered an error in our evaluation after the submission of the final version of the paper. In the wrong version, we computed average values also from results obtained when different noise placement heuristics than random-based were enabled.

Table 4.4: Average relative degradation of execution time when using different types of noise (timeoutTamper and haltOneThread disabled, seqLen=50)

		Noise Type						
Test	nFreq	yield	sYield	sleep	wait	bWait	mixed	aver.
Sunbank	50	**1.00**	**1.00**	1.07	1.42	4.92	1.07	1.75
	100	**1.00**	1.01	1.24	1.86	9.14	1.27	2.59
	150	**1.00**	1.02	1.82	2.28	13.05	1.82	3.50
	200	**1.00**	**1.00**	2.29	2.72	17.10	2.28	4.40
Airlines	50	0.99	**0.98**	2.76	3.27	17.11	2.55	4.61
	100	**0.99**	1.00	4.83	5.73	31.47	4.48	8.08
	150	1.00	**0.99**	6.48	7.67	46.73	5.91	11.46
	200	**0.99**	1.00	8.74	11.01	62.65	7.57	15.33
Crawler	50	**0.95**	0.98	1.15	1.11	1.81	1.13	1.19
	100	1.00	**0.93**	1.16	1.24	2.43	1.15	1.32
	150	**1.04**	**1.04**	1.30	1.44	3.05	1.28	1.52
	200	1.03	**0.96**	1.46	1.46	3.71	1.39	1.67
FTPServer	50	**0.94**	1.16	1.68	1.27	2.13	1.44	1.44
	100	**0.92**	0.94	1.72	1.49	1.38	1.76	1.37
	150	1.03	**0.93**	1.75	1.66	1.77	1.89	1.51
	200	**1.02**	1.05	1.58	1.44	1.02	1.45	1.26
TIDOrbJ	50	**0.99**	1.00	1.56	1.41	5.02	1.40	1.90
	100	**0.99**	1.00	2.11	2.00	9.18	1.82	2.85
	150	**1.00**	1.01	2.67	2.68	9.63	2.27	3.21
	200	**1.00**	1.02	3.22	3.38	9.63	2.73	3.49

The table illustrates that noise injection affects each of the considered test cases differently—sometimes it helps, sometimes not. The use of noise almost always very significantly helps in the cases of Sunbank and Airlines, but it does not help in the case of FTPServer. Also, the different seeding techniques perform differently in the different test cases, and one cannot claim a clear winner among them. The *wait* technique helps the most in the Sunbank test case while *yield* provides the best improvement in the Airlines test case. In the case of FTPServer, no technique provides any improvement. Moreover, the *synchYield* technique with *nFreq* set to higher values than 50 totally suppressed manifestation of the error in the Sunbank test case. A significant influence of *nFreq* is visible in the Sunbank test case, but in the FTPServer case, it seems that *nFreq* has no influence. The effect of *nFreq* in Airlines has no clear tendency. Nevertheless, overall, the table demonstrates that choosing a suitable noise seeding technique can rapidly improve the probability of detecting an error at least in some cases.

Table 4.2 shows the average relative improvement of the Avio* coverage metrics. Again, the influence of the different noise seeding techniques varies for different test cases. The average improvement is in general smaller than the improvement in error manifestation presented above. Most of noise techniques on average help in the Crawler and FTPServer test cases only. The *yield* technique was often the only heuristics that helps in the Sunbank and Airlines test cases. Most of the noise techniques provide considerably worse results for the Airlines (namely, *sleep*, *wait*, *busyWait*, and *mixed* techniques). Finally, no technique helps to improve the Avio* coverage in the TIDOrbJ test case. The influence of the *nFreq* parameter can be clearly seen only in the smallest test case—the Sunbank.

In Table 4.3, the same comparison using the HBPair* coverage metrics is provided. The Sunbank test case is omitted because even the configuration which disables noise achieved the full coverage, and therefore other configurations cannot provide any improvement. Again, noise techniques help in the Crawler and FTPServer test cases, and one can see the same champions providing the best improvement in the considered test cases and noise frequencies. Most of the noise techniques provide considerably worse results in the Airlines test case, and the *nFreq* parameter has no clear influence again.

Overall, Tables 4.2 and 4.3 show that choosing a suitable technique can improve the Avio* and HBPair* coverage metrics. In the Sunbank and Airlines test cases, the highest coverage improvement was achieved using the *yield* technique which provides a poor or no improvement in the Crawler and FTPServer test cases. In these test cases, the *busyWait* and *mixed* techniques often provide the best coverage improvement. The low or none improvement in the TIDOrbJ test case indicates that random noise placement heuristics is not able to improve the coverage in this test case.

Finally, Table 4.4 provides a comparison of the noise techniques according to their influence on the execution time of tests. The table shows the average relative degradation of the execution time that we observed when using different basic noise seeding techniques. The entries of the table therefore give the ratio of the execution time observed when using noise injection of the respective type against the execution time without any noise setting enabled. Lower values mean that noise of the appropriate type provides lower degradation of execution time. The value 1.00 means that the given type of noise on average provides no time degradation at all.

In Table 4.4, one can see the average effect (in the *aver.* column) of the *nFreq* parameter which controls the amount of noise injected into the execution in all test cases but the FTPServer where the increasing tendency of execution time degradation with the increasing value of the *nFreq* parameter can be seen only on fragments of results. This is probably caused by the nature of the test case which uses a timer to invoke the server shutdown sequence (and hence to begin to refuse new client connections) after a predefined time. The active connections are served and therefore we can see different time degradation values which are, however, influenced by the num-

Table 4.5: Influence of noise frequency (nFreq) and noise type on the best error manifestation improvement (seqLen=50)

		Noise Type						
Test	nFreq	yield	sYield	sleep	wait	bWait	mixed	aver.
Sunbank	50	4.87	0.93	**5.93**	5.60	2.60	1.70	3.61
	100	5.33	0.97	5.90	**8.27**	4.87	6.97	5.38
	150	**9.50**	0.83	8.23	4.97	4.07	4.97	5.43
	200	8.73	0.97	7.60	**12.07**	6.13	7.47	7.16
Airlines	50	**3.45**	2.71	2.38	2.79	2.25	3.27	2.81
	100	**5.57**	3.96	3.18	2.25	2.95	2.70	3.43
	150	**4.41**	4.27	1.45	2.21	2.27	2.23	2.81
	200	**5.09**	2.70	1.80	2.82	1.71	2.29	2.74
FTPServer	50	**0.97**	0.93	0.54	0.59	0.62	0.62	0.71
	100	**1.02**	0.94	0.37	0.40	0.58	0.48	0.63
	150	**1.02**	**1.02**	0.42	0.40	0.71	0.46	0.67
	200	0.97	**0.99**	0.34	0.33	0.61	0.53	0.63

ber of active client connections at the moment of invoking the shutdown sequence. Surprisingly, the table shows that in a few cases, the *yield* and *synchYield* techniques provide even a little improvement to the configuration with disabled noise. This is probably caused by lucky scheduling scenarios in which the threads are not blocked before entering a critical sections so often. Overall, one can see that some techniques (namely, the *yield* and *synchYield*) degrade the execution time only a little while other techniques generate a considerable execution time degradation (mainly the *busyWait* technique) which is influenced by the *nFreq* parameter and can achieve quite high values.

Influence of the Advanced Noise Seeding Techniques

This section presents results which show that a suitable combination of basic and advanced noise seeding techniques can in some cases rapidly improve the testing process. The goal of the comparison presented here is to study the positive effect of advanced noise seeding techniques. The results can be used to infer a suitable setting of the noise seeding parameters in order to maximise the considered metric. The results are summarised in 7 tables.

Tables 4.5, 4.6, and 4.7 show the maximal improvement achieved when using the different basic noise seeding techniques and different values of the *nFreq* parameter combined with both considered advanced noise seeding techniques. Each cell in the tables therefore contains the best improvement of the error detection ratio, Avio* cov-

Table 4.6: Influence of noise frequency (nFreq) and noise type on the best Avio* coverage improvement

		Noise Type						
Test	nFreq	yield	sYield	sleep	wait	bWait	mixed	aver.
Sunbank	50	1.21	1.26	1.16	1.16	1.09	**1.28**	1.19
	100	1.21	1.15	1.02	0.97	0.92	**1.29**	1.09
	150	1.21	**1.42**	1.01	1.02	1.00	1.22	1.15
	200	1.22	**1.29**	1.02	1.02	1.00	1.02	1.09
Airlines	50	**1.99**	0.90	1.05	0.95	0.84	0.97	1.12
	100	**1.46**	0.51	0.88	0.97	0.77	0.93	0.92
	150	**1.79**	0.95	0.94	0.88	0.94	0.84	1.06
	200	**1.56**	0.96	0.92	0.85	0.90	0.97	1.03
Crawler	50	6.65	**7.87**	6.42	6.37	5.53	6.99	6.64
	100	6.58	6.74	**7.20**	6.00	5.73	6.71	6.49
	150	6.87	6.49	6.84	6.71	3.75	**7.05**	6.28
	200	7.26	**7.42**	6.44	5.88	4.12	6.33	6.24
FTPServer	50	1.17	1.11	**1.24**	1.22	1.20	1.21	1.19
	100	1.15	1.11	**1.22**	**1.22**	1.19	1.20	1.18
	150	1.18	1.10	**1.23**	1.21	1.13	1.19	1.17
	200	1.13	1.15	1.21	**1.22**	1.11	1.20	1.17
TIDOrbJ	50	**1.02**	1.01	0.97	0.96	1.00	0.98	0.99
	100	0.99	**1.00**	0.96	0.95	0.97	0.98	0.98
	150	1.00	**1.02**	0.96	0.92	0.88	0.99	0.96
	200	**1.03**	1.02	0.93	0.96	0.94	0.99	0.98

erage, and HBPair* coverage, respectively, taken from the 4 possible configurations (random noise of the appropriate type and frequency, random noise with *timeoutTampering* enabled, random noise with *haltOneThread* enabled, and random noise with both *timeoutTampering* and *haltOneThread* enabled).

The next three tables, i.e. Tables 4.8, 4.9, and 4.10, further study the influence of the *timeoutTampering* and *haltOneThread* noise seeding heuristics on the considered test cases and metrics. The tables contain the best relative improvement values from the 4 possible configurations (random noise of appropriate type with the noise frequency set to 50, 100, 150, and 200).

Finally, Table 4.11 shows the average influence of the advanced noise seeding techniques on the execution time. All tables present results for *SeqLen*=50. We will now discuss the results in more detail.

Table 4.5 shows the influence of the *noise frequency* (nFreq) and the *noise type* on error detection. Moreover, average values (denoted as aver. in the table) are pro-

Table 4.7: Influence of noise frequency (nFreq) and noise type on the best HBPair* coverage improvement

Test	nFreq	Noise Type						aver.
		yield	sYield	sleep	wait	bWait	mixed	
Airlines	50	**1.90**	0.89	0.94	0.93	0.90	0.94	1.08
	100	**1.49**	0.52	0.87	0.95	0.86	0.92	0.94
	150	**1.81**	0.97	0.87	0.86	0.85	0.84	1.03
	200	**1.54**	0.95	0.87	0.84	0.85	0.91	1.00
Crawler	50	3.18	3.31	3.38	3.29	3.26	**3.51**	3.32
	100	3.28	3.17	**3.42**	3.19	2.93	3.41	3.23
	150	3.29	3.13	3.27	3.27	2.38	**3.34**	3.11
	200	3.32	**3.34**	3.33	3.07	2.43	3.17	3.11
FTPServer	50	1.23	1.23	1.30	1.25	**1.40**	1.25	1.28
	100	1.25	1.22	**1.51**	1.40	1.44	1.25	1.34
	150	1.25	1.23	1.45	1.30	**1.48**	1.25	1.33
	200	1.30	1.24	1.41	1.40	**1.47**	1.32	1.36
TIDOrbJ	50	0.99	1.00	**1.05**	0.98	1.04	0.98	1.01
	100	1.01	0.98	**1.05**	0.99	1.03	0.99	1.01
	150	0.95	0.98	**1.08**	0.92	0.99	1.07	1.00
	200	1.06	1.03	**1.09**	1.04	1.05	1.05	1.05

vided for a better comparison. It can be seen that different noise types and noise frequencies influence the considered test cases differently. For instance, in the Airlines and Sunbank test cases, all noise techniques provide a striking improvement, but in the FTPServer test case, the maximal improvement is rather small reaching one or two percents. In the Sunbank test case, the *synchYield* (denoted as sYield) heuristics provides poor results, and the *wait* heuristics with the noise frequency set to 200, provides the best overall improvement. In the Airlines test case, the best results were obtained when the *yield* noise seeding technique was used. The FTPServer test shows that nearly all configurations achieved poor or none improvement. This indicates that the random noise placement heuristics is able to improve the error manifestation ratio only minimally with the considered noise seeding techniques .

The Crawler test case is not listed because the error was not detected without noise. The error was detected only when the *busyWait* noise seeding technique was enabled and the noise frequency set to 150 or 200. The TIDOrbJ test case is not listed because it does not contain any error.

When comparing results presented in Table 4.5 with similar Table 4.1 which presents the improvement obtained by basic noise seeding techniques only, one can see that the combination of basic and advanced noise seeding techniques leads to a considerable improvement in some cases. For instance, the improvement in the Sunbank test case for the *yield* technique ranged from 0.82 to 5.28 when only basic noise injection techniques were used and 4.87 to 9.50 when a suitable combination with the advanced noise seeding techniques is considered. Also, we were not able to detect the error in Crawler at all when using the basic noise seeding techniques while we were able to detect it when using the *timeoutTampering* heuristics.

On the other hand, the combination with the advanced noise seeding techniques does help only a little in some cases. For instance, in the Sunbank test case, and for the *busyWait* technique, almost all the results remain the same. The only improvement was achieved for *nFreq*=150 where the value increased from 3.27 in Table 4.1 to 4.07 in Table 4.5. In the FTPServer test case, the improvement is also none or minimal.

Table 4.6 shows results for the maximal improvement of the Avio* coverage metric. The results again vary for each test case. This time, the noise frequency seems to have minimal influence on the result. The improvement in the Crawler test case is, however, very high. The overall maximal improvement for the Crawler test case when only the basic noise seeding techniques were used (presented in Table 4.2) was 2.05. The maximal improvement with the advanced noise seeding techniques reached 7.87. The values for other configurations are also much higher. This is caused by the positive effect of the *timeoutTamper* heuristics on this test case as explained below. A positive effect of the advanced noise seeding techniques can also be observed in the Sunbank, Airlines, and FTPServer test cases. In the Airlines test case, the best results were achieved using *yield* similarly as in Table 4.2. On the other hand, the best results were achieved with different basic noise seeding techniques in the case of Sunbank. Only minimal or none improvement can be seen in the TIDOrbJ test case.

Table 4.7 shows results for the maximal improvement of the HBPair* coverage metric. The Sunbank test case is missing in the table. The synchronisation in the test case is simple and therefore all configurations including the configuration which disables noise reached the full coverage. For the Airlines test case, the table shows a minimal improvement except the column showing results for the *yield* heuristics. A minimal improvement was also achieved in the TIDOrbJ test case. The noise injection helps to improve the HBPair* coverage in the Crawler and FTPServer test cases. In comparison with Table 4.2, one can clearly see the positive effect of advanced noise seeding techniques on the Crawler test case where the improvement is on average almost 3 times higher than when the advanced noise seeding techniques are not used. A smaller improvement can be seen in all other considered test cases.

Tables 4.8, 4.9, and 4.10 study the influence of the *timeoutTampering* and *haltOneThread* noise seeding heuristics on the considered test cases and metrics in more detail. The tables contain the best relative improvement from the 4 possible configurations (random noise of the given type with the noise frequency set to 50, 100, 150, and 200).

Table 4.8: Influence of the timeoutTampering (tt_) and haltOneThread (ht_) on the best error detection improvement

		tt_0		tt_1	
Test	Noise type	ht_0	ht_1	ht_0	ht_1
Sunbank	yields	**9.50**	4.87	7.90	6.20
	sYield	0.87	**0.97**	0.80	0.83
	sleep	**8.23**	2.37	7.47	6.67
	wait	**12.07**	2.37	8.17	3.33
	bWait	**6.13**	3.97	3.30	4.07
	mixed	6.97	2.57	**7.47**	2.50
Airlines	yields	5.09	2.20	3.45	**5.57**
	sYield	2.66	2.82	2.29	**4.27**
	sleep	1.88	2.23	**3.18**	1.75
	wait	1.68	**2.82**	2.79	1.91
	bWait	2.25	2.18	1.93	**2.95**
	mixed	2.70	2.75	2.23	**3.27**
FTPServer	yields	**1.02**	0.99	0.79	0.84
	sYield	**1.02**	0.99	0.79	0.74
	sleep	**1.01**	0.54	0.28	0.37
	wait	**1.01**	0.59	0.41	0.35
	bWait	**1.01**	0.71	0.58	0.62
	mixed	**1.01**	0.62	0.46	0.61

Table 4.8 shows the influence of the *timeoutTampering* (denoted as *tt*, *tt_0* means that the heuristics was disabled and *tt_1* means that the heuristics was enabled) and *haltOneThread* (denoted as *ht*) noise seeding heuristics and their combination with the primitive noise seeding heuristics (denoted as *Noise type* in the table) on the error detection improvement. The values in the tables correspond to values in Tables 4.1 and 4.5 and represent the last piece of the puzzle providing a complex view on the comparison of noise seeding techniques and their combinations. In particular, the column which shows the best results for the case when both advanced noise seeding techniques were disabled (the column denoted as tt_0 and ht_0) presents the best values for the particular test case and noise type taken from Table 4.1. The best value in each row (highlighted in bold) represents a value which provided the best improvement and therefore appeared in Table 4.5. Therefore, by combining these three tables, one can determine for each test case the best combination of noise seeding techniques and study the influence of all parameters on average improvement of the error detection ratio as we discussed below.

Table 4.9: Influence of the timeoutTampering (tt_) and haltOneThread (ht_) on the best Avio* improvement

		tt_0		tt_1	
Test	Noise type	ht_0	ht_1	ht_0	ht_1
Sunbank	yields	**1.22**	1.12	**1.22**	1.21
	sYield	1.06	1.01	1.29	**1.42**
	sleep	1.02	**1.16**	1.01	1.02
	wait	1.02	0.91	1.02	**1.16**
	bWait	1.01	**1.09**	0.92	0.90
	mixed	1.02	1.23	1.00	**1.29**
Airlines	yields	**1.03**	0.88	0.94	1.00
	sYield	**1.03**	0.51	0.89	0.82
	sleep	1.03	0.94	**1.05**	0.88
	wait	**1.03**	0.90	0.88	0.97
	bWait	**1.03**	0.94	0.51	0.90
	mixed	**1.03**	0.87	0.97	0.97
Crawler	yields	1.00	1.27	7.21	**7.26**
	sYield	1.00	1.00	7.42	**7.87**
	sleep	1.64	1.54	6.37	**7.20**
	wait	1.43	1.66	**6.71**	6.49
	bWait	2.05	3.92	**5.73**	5.53
	mixed	1.37	3.73	**7.05**	6.66
FTPServer	yields	1.04	1.07	1.15	**1.18**
	sYield	1.06	1.07	**1.15**	**1.15**
	sleep	1.19	1.23	1.16	**1.24**
	wait	1.14	**1.22**	**1.22**	1.21
	bWait	1.19	1.19	**1.20**	1.19
	mixed	1.20	1.20	1.20	**1.21**
TIDOrbJ	yields	1.00	**1.03**	0.99	1.02
	sYield	**1.02**	**1.02**	1.01	**1.02**
	sleep	**1.00**	0.96	0.92	0.97
	wait	**1.00**	0.96	0.94	0.96
	bWait	**1.00**	**1.00**	0.97	**1.00**
	mixed	**1.00**	0.98	0.94	0.99

For instance, the best improvement in the Sunbank test case is 12.07. The occurrence of this value in the row denoted tt_0 and ht_0 indicates that the value was obtained when both advanced noise seeding techniques were disabled. Table 4.1 shows us that the value was achieved using the noise frequency 200 and the *wait*

Table 4.10: Influence of the timeoutTampering (tt_) and haltOneThread (ht_) on the best HBPair* improvement

		tt_0		tt_1	
Test	Noise type	ht_0	ht_1	ht_0	ht_1
Airlines	yields	**1.02**	0.89	0.93	0.97
	sYield	**1.02**	0.52	0.87	0.86
	sleep	**1.02**	0.88	0.94	0.86
	wait	**1.02**	0.93	0.85	0.95
	bWait	**1.02**	0.90	0.51	0.86
	mixed	**1.02**	0.90	0.94	0.92
Crawler	yields	1.03	1.07	3.26	**3.32**
	sYield	1.13	1.04	**3.34**	3.31
	sleep	1.17	1.25	3.33	**3.42**
	wait	1.17	1.19	**3.29**	3.26
	bWait	1.49	2.69	2.93	**3.26**
	mixed	1.18	2.27	3.49	**3.51**
FTPServer	yields	1.12	1.21	1.24	**1.30**
	sYield	1.16	1.15	**1.24**	1.23
	sleep	1.21	1.51	1.25	**1.43**
	wait	1.25	**1.40**	1.25	**1.40**
	bWait	1.44	1.44	1.30	**1.48**
	mixed	1.25	1.25	1.25	**1.32**
TIDOrbJ	yields	1.00	**1.06**	0.98	1.00
	sYield	**1.03**	1.00	1.00	0.99
	sleep	1.00	**1.09**	0.98	1.05
	wait	1.00	**1.04**	0.95	1.00
	bWait	1.00	1.03	0.99	**1.05**
	mixed	1.00	0.99	0.97	**1.07**

seeding technique. The table also shows how much worse the other configurations of basic noise seeding techniques were. Table 4.5, one can see how much the combination of particular basic noise seeding techniques with the *timeoutTampering* and *haltOneThread* techniques differ from the overall best value 12.07.

Table 4.8 shows that in the case of FTPServer, the best improvement was obtained when both heuristics were disabled. Using either of the heuristics actually leads to significantly worse results. As for the Sunbank and Airlines test cases, no clear tendency can be identified. The results for Crawler and TIDOrbJ test cases are omitted. In the Crawler test case, the error was detected when both heuristics were disabled and also when *timeoutTampering* heuristics was enabled and *haltOneThread* heuris-

Table 4.11: Influence of the *haltOneThread* (ht_) and *timeoutTamper* (tt_) techniques on time degradation (seqLen=50)

	tt_0		tt_1	
Test	ht_0	ht_1	ht_0	ht_1
Sunbank	**2.65**	3.55	2.96	3.45
Airlines	**8.10**	12.61	9.82	12.50
Crawler	1.41	1.60	**1.23**	1.36
FTPServer	1.38	**1.29**	1.54	1.45
TIDOrbJ	**2.79**	3.54	2.86	3.54

tics disabled. Results for the TIDOrbJ test case are missing because the test case contains no error.

Tables 4.9 and 4.10 show influence of the considered heuristics on the Avio* and HBPair* coverage improvement. The tables refine values presented in Tables 4.6 and 4.7, respectively. In general, no clear improvement provided by the considered heuristics can be seen with one exception. In the Crawler test case, the positive effect of the *timeoutTampering* heuristics is evident. The test contains many timed routines which model the environment. The *timeoutTampering* heuristics influences these timeouts in a way leading to significantly better results in both the Avio* and HBPair* coverage metrics as well as in the error manifestation ratio.

Finally, Table 4.11 shows influence of the advanced noise seeding techniques on the execution time. The table shows the average relative degradation of the execution time that we observed when using different advanced noise seeding techniques. Each cell therefore contains an average value computed from 24 values (6 basic noise heuristics applied with 4 different noise frequencies). Lower values mean that noise of the appropriate type provides lower degradation of execution time. Naturally, using additional noise seeding techniques increases the time degradation. There are two exceptions in the FTPServer and Crawler test cases where the use of *haltOneThread* in the FTPServer test case and *timeoutTampering* in the Crawler test case had a little positive effect. The *haltOneThread* technique in the FTPServer test case probably stops one of the threads that serve client requests and therefore other client threads can proceed a bit faster. The *timeoutTampering* technique has a positive effect in the Crawler test case because the technique sometimes makes the affected timeout shorter than the original timeout used in the code. Overall, the time degradation introduced by *haltOneThread* seems to be a bit higher than the execution time degradation caused by *timeoutTampering*.

Overall, the results presented in this section indicate that there is no optimal configuration, and for each test case and each testing goal, one needs to choose a different testing configuration. In some cases, the random noise placement heuristics does not provide any improvement. Moreover, some noise seeding configurations used with the random noise placement heuristics actually provide considerably worse results than the configuration with disabled noise seeding.

4.3.3 A Comparison of Noise Placement Heuristics

This section concentrates on the influence of the ConTest's heuristics restricting noise generation to events related to shared variables and on the influence of our new heuristics proposed in Section 4.2. In addition, the scenario in which ConTest randomly chooses its own parameters and the scenarios in which ConTest advanced noise seeding techniques are enabled and disabled are also considered. The comparison is divided into two parts: (i) First, the comparison using total values is provided. In this comparison, the noise placement heuristics are compared according to the error manifestation ratio and the cumulative coverage they achieved on average. Therefore, the comparison favours techniques which provide a high coverage. (ii) Then, a comparison which compares noise placement heuristics according to total values divided by the execution time is provided. Intuitively, this comparison favours techniques that provide a high coverage with a low overhead, and therefore punishes techniques that either put too much noise into test executions or achieve a poor coverage only.

Each part of the comparison contains three tables comparing the noise placement heuristics using the error manifestation ratio, the Avio* coverage, and the HBPair* coverage. The *configuration* column of all tables describes the considered noise injection configuration. A configuration consists of five parts delimited by the "_" character. The meaning of these parts is as follows: (Part 1) The ConTest *random* parameter: If set to 1, ConTest parameters considered in Parts 2–4 are set randomly before each execution. (Part 2) If set to 1, the *timeoutTamper* heuristics is enabled. (Part 3) If set to 1, the *haltOneThread* heuristics is enabled. (Part 4) This part is divided into three sub-parts delimited by "-". The first sub-part indicates whether the ConTest's heuristics limiting noise generation to events related to shared variables is enabled. The second sub-part says whether the noise is also put to other *ploc*s than accesses to shared variables. Finally, the third sub-part says whether the noise is put to *all* shared variables or *one* randomly chosen before each execution. (Part 5) This last part encodes the setting of our noise injection heuristics. It consists of two sub-parts delimited by "-". The first sub-part says whether our noise injection heuristics is enabled and the second one whether our noise strength computation is enabled too.

For each considered test case (i.e., Airlines, Crawler, etc.), the test configurations were ranked according to the obtained results—rank 1 is the best, rank 23 is the worst. More precisely, the entries of the tables under the particular test cases contain average ranks obtained across the different basic noise types of ConTest. The average rank over all the test cases is provided in the last column (denoted as *aver.*). The test configurations are then sorted according to their average rank, giving their final position in the evaluation of the 23 configurations. The final position (denoted as *Pos.*) is used to identify the configurations in the following text. For a better overview, the rank achieved by the best ranked configuration for each test case is highlighted in bold.

A Comparison Using *total* Values

In this subsection, the noise placement heuristics are compared without considering the overhead they introduce. The results are summarised in Tables 4.12, 4.13, and 4.14 showing ranking of the heuristics according to the error manifestation ratio, Avio*, and HBPair* coverage respectively.

Table 4.12 presents a comparison using the error manifestation ratio. The table shows that the efficiency of different configurations of the heuristics vary for different test cases. There is no configuration which wins in more than one test case. The overall best configuration (at position 1) does even not provide the best result in any of the test cases. Our noise heuristics with the newly proposed noise strength computation (at position 4) achieved much better results than the version of our heuristics without this feature (at position 11) which was, however, able to detect the error in the Crawler test case. There were only three configurations which were able to detect the error in the Crawler test case. Surprisingly, good results were achieved using the ConTest random setting (at position 5). The configuration even achieved the best rank in the Airlines test case.

In general, one can see that the first 11 positions are occupied by various combinations of the heuristics which focus on a single variable (having *one* within the configuration string), both of our newly proposed heuristics at positions 4 and 11, and the random setting of ConTest at position 5. The configuration which uses only random noise placement heuristics without any advanced noise seeding heuristics is at position 14. This indicate that the improvement presented in Table 4.1 to compare noise seeding techniques was not the most successful heuristics. Good results provided by the heuristics which focus on a single variable prove conclusions presented in [16] that focusing noise on a single variable that is randomly chosen for each test execution improves the efficiency of the testing process.

Table 4.13 shows results of a comparison using the Avio* coverage. In this case, not all of the configurations which focus the noise on a single variable are ranked high. However, some configurations with this noise placement heuristics again achieved the overall best ranks. The best rank in the FTPServer was achieved by the configuration which combines basic and both advanced noise seeding techniques with a noise focused on all shared variables (at position 5).

Our heuristics with the newly proposed noise strength computation (at position 9) achieved the best rank in the biggest test case which is the TIDOrbJ. The value 1.0 means that the heuristics provides the highest coverage regardless which of the noise

Table 4.12: A comparison of heuristics using *total* values of error manifestation results (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuraiton	Sunbank	Airlines	Crawler	FTPServer	aver.
1	0_0_1_1-0-one_0-0	4.8	7.2	11.2	3.0	6.6
2	0_0_0_1-0-one_0-0	8.7	9.5	11.2	**2.7**	8.0
3	0_0_1_1-1-one_0-0	4.7	9.2	11.2	8.0	8.3
4	0_0_0_0-0-all_1-1	9.7	8.3	11.3	3.7	8.3
5	1_0_0_0-0-all_0-0	14.0	**1.7**	11.2	7.8	8.7
6	0_1_0_1-0-one_0-0	6.2	8.0	11.2	10.8	9.1
7	0_1_1_1-0-one_0-0	7.0	10.5	11.2	9.5	9.6
8	0_1_1_1-1-one_0-0	**3.2**	10.8	11.2	13.3	9.6
9	0_1_0_1-1-one_0-0	6.5	11.7	11.2	11.8	10.3
10	0_0_0_1-1-one_0-0	7.7	18.0	11.2	5.8	10.7
11	0_0_0_0-0-all_0-1	9.8	15.6	**9.3**	8.0	10.7
12	0_0_1_1-0-all_0-0	13.5	11.3	9.5	13.5	12.0
13	0_0_1_0-0-all_0-0	20.0	9.7	11.2	9.7	12.7
14	0_0_0_0-0-all_0-0	18.7	14.8	11.2	7.3	13.0
15	0_0_0_1-1-all_0-0	12.5	15.3	11.2	13.3	13.1
16	0_1_1_0-0-all_0-0	15.3	8.3	11.2	18.0	13.2
17	0_1_1_1-0-all_0-0	17.7	11.8	**9.3**	15.7	13.6
18	0_0_0_1-0-all_0-0	13.0	15.3	11.2	15.2	13.7
19	0_1_1_1-1-all_0-0	15.3	12.5	11.2	16.0	13.8
20	0_1_0_1-1-all_0-0	9.7	16.5	11.2	18.5	14.0
21	0_1_0_1-0-all_0-0	13.8	12.2	11.2	20.2	14.4
22	0_0_1_1-1-all_0-0	19.2	14.5	11.2	13.3	14.6
23	0_1_0_0-0-all_0-0	14.3	11.8	11.2	21.2	14.6

seeding heuristics was used. The version of our heuristics without the noise strength computation (at position 16) provides almost always the second best result in this test case.

The table also shows that the configuration which uses random noise placement heuristics only (at position 21) and configurations which focus all variables and which do not use any advanced noise seeding heuristics (at positions 22 and 23) are not able to achieve a high coverage. This can be considered as a demonstration that inserting noise in too many places or at random does not necessarily improve the testing process of concurrent programs.

Finally, Table 4.14 presents results for the HBPair* coverage. The Sunbank test case is omitted because all configurations achieved the full coverage of this metric

Table 4.13: A comparison of heuristics using *total* values of Avio* coverage results (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuraiton	Sunb.	Airl.	Craw.	FTPS.	TIDO.	aver.
1	0_1_1_1-1-one_0-0	4.3	**4.2**	8.5	10.5	3.0	6.1
2	0_1_1_1-0-one_0-0	**2.5**	8.0	5.8	11.3	10.7	7.7
3	0_1_0_1-1-one_0-0	3.7	8.0	7.7	12.2	9.7	8.3
4	0_1_0_1-0-one_0-0	6.7	6.3	4.2	11.8	13.0	8.4
5	0_1_1_1-1-all_0-0	17.0	9.3	6.8	**3.8**	9.7	9.3
6	0_0_1_1-1-one_0-0	5.3	6.3	15.3	17.5	4.3	9.7
7	1_0_0_0-0-all_0-0	10.0	17.7	**1.5**	13.7	6.7	9.9
8	0_1_1_1-0-all_0-0	16.3	14.8	4.7	2.0	11.8	9.9
9	0_0_0_0-0-all_1-1	10.7	9.3	11.7	19.0	**1.0**	10.3
10	0_0_0_1-1-one_0-0	3.2	6.7	16.7	18.5	7.8	10.6
11	0_1_1_0-0-all_0-0	14.0	12.3	7.8	7.5	12.8	10.9
12	0_0_1_1-1-all_0-0	17.2	10.2	14.5	7.7	8.8	11.7
13	0_0_1_1-0-one_0-0	6.3	5.7	19.7	15.2	12.7	11.9
14	0_0_1_1-0-all_0-0	16.5	11.7	15.2	5.5	14.2	12.6
15	0_1_0_1-0-all_0-0	17.5	16.2	4.5	7.2	18.5	12.8
16	0_0_0_0-0-all_0-1	10.8	12.2	19.3	20.0	2.5	13.0
17	0_1_0_1-1-all_0-0	13.0	18.5	6.5	8.2	20.3	13.3
18	0_1_0_0-0-all_0-0	14.7	15.8	7.7	9.8	18.7	13.3
19	0_0_0_1-0-one_0-0	6.0	11.3	19.8	18.8	13.7	13.9
20	0_0_1_0-0-all_0-0	17.2	14.0	15.2	8.3	15.2	14.0
21	0_0_0_0-0-all_0-0	14.5	8.8	18.7	19.8	13.2	15.0
22	0_0_0_1-1-all_0-0	16.0	15.7	16.0	14.2	16.0	15.6
23	0_0_0_1-0-all_0-0	16.8	18.3	18.3	11.0	20.2	16.9

in this test case. Positions of the best ranked configurations for each test case (e.g., configuration at position 15 which provided the best rank for the Crawler test case) indicate that the configuration suitable for one test case provided poor results in the other test cases.

Our heuristics with the newly proposed noise strength computation achieved the best rank for the TIDOrbJ test case again. Two out of four best ranks were achieved by the configurations which focus noise on all shared variables (namely, configurations at positions 4 and 15). However, the configurations which focus noise on a single shared variable still achieve very good results in the overall ranking. The ConTest random setting (at position 11) achieved worse results than in the previous comparisons. Unsurprisingly, the configuration which uses only random noise (at position 22) provide poor results except the Airlines test case which uses a rather simple synchronisation.

Table 4.14: A comparison of heuristics using *total* values of HBPair* results (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuraiton	Airlines	Crawler	FTPServer	TIDOrbJ	aver.
1	0_1_1_1-1-one_0-0	5.0	5.8	8.0	3.3	5.5
2	0_1_1_1-0-one_0-0	6.7	5.3	7.2	8.7	7.0
3	0_1_0_1-1-one_0-0	8.0	6.7	7.3	8.8	7.7
4	0_1_1_1-1-all_0-0	9.2	7.3	**2.7**	12.2	7.9
5	0_1_0_1-0-one_0-0	6.2	3.8	13.3	11.5	8.7
6	0_0_1_1-1-one_0-0	5.8	15.8	10.8	3.8	9.1
7	0_1_1_1-0-all_0-0	15.2	4.8	5.8	11.2	9.3
8	0_0_1_1-1-all_0-0	10.8	14.3	6.5	7.7	9.8
9	0_1_1_0-0-all_0-0	13.7	7.7	5.5	14.3	10.3
10	0_0_0_0-0-all_1-1	9.0	11.7	17.7	**3.0**	10.4
11	1_0_0_0-0-all_0-0	17.0	7.2	10.2	10.2	11.2
12	0_0_1_1-0-all_0-0	12.3	14.8	8.7	9.7	11.4
13	0_0_1_1-0-one_0-0	**4.0**	20.3	12.5	10.5	11.8
14	0_0_0_1-1-one_0-0	7.3	17.3	13.2	10.8	12.2
15	0_1_0_1-0-all_0-0	15.7	**2.7**	15.2	17.3	12.7
16	0_0_1_0-0-all_0-0	15.7	16.0	9.3	12.7	13.4
17	0_1_0_1-1-all_0-0	19.2	7.8	11.7	16.7	13.9
18	0_1_0_0-0-all_0-0	16.7	7.0	15.0	18.2	14.2
19	0_0_0_1-0-one_0-0	11.7	19.7	16.0	11.8	14.8
20	0_0_0_1-1-all_0-0	14.5	16.5	13.5	16.3	15.2
21	0_0_0_0-0-all_0-1	10.8	19.5	19.8	9.5	14.9
22	0_0_0_0-0-all_0-0	9.0	18.5	21.0	15.8	16.1
23	0_0_0_1-0-all_0-0	20.2	19.0	14.7	16.2	17.5

A Comparison Using *total/time* Values

In this subsection, a comparison of the noise placement heuristics which considers both the obtained coverage and the time degradation introduced by the noise is presented. The results are summarised in three tables. In particular, Tables 4.15, 4.16, and 4.17 show results for the error manifestation ratio, the Avio* coverage, and the HBPair* coverage, respectively.

In Table 4.15, a comparison using the error manifestation ratio is presented. The TIDOorbJ test case is missing because it does not contain any error. The table shows that the efficiency of different configurations of heuristics vary for different test cases again. There is no configuration which wins in more than one test case, and the

Table 4.15: A comparison of heuristics using *total/time* and error manifestation probability (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuraiton	Sunbank	Airlines	Crawler	FTPServer	aver.
1	0_0_1_1-0-one_0-0	4.8	7.7	11.2	2.3	6.5
2	0_0_0_0-0-all_0-1	5.4	8.8	**9.3**	4.7	7.1
3	0_0_0_1-0-one_0-0	8.3	7.7	11.2	**2.0**	7.3
4	1_0_0_0-0-all_0-0	9.7	**1.5**	11.2	7.7	7.5
5	0_0_0_0-0-all_1-1	6.0	5.0	11.3	9.3	7.9
6	0_0_1_1-1-one_0-0	6.2	7.8	11.2	8.5	8.4
7	0_1_0_1-0-one_0-0	5.3	7.8	11.2	10.7	8.8
8	0_1_1_1-0-one_0-0	6.0	8.3	11.2	10.2	8.9
9	0_1_0_1-1-one_0-0	4.8	10.3	11.2	12.2	9.6
10	0_1_1_1-1-one_0-0	**3.3**	9.5	11.2	14.7	9.7
11	0_0_0_1-1-one_0-0	8.0	13.7	11.2	7.2	10.0
12	0_0_0_0-0-all_0-0	18.3	9.8	11.2	6.0	11.3
13	0_0_1_1-0-all_0-0	15.3	13.3	9.5	11.8	12.5
14	0_0_0_1-1-all_0-0	14.5	16.2	11.2	13.3	13.8
15	0_0_0_1-0-all_0-0	15.3	14.8	11.2	14.7	14.0
16	0_0_1_0-0-all_0-0	20.7	14.8	11.2	10.0	14.2
17	0_1_0_1-0-all_0-0	14.5	13.3	11.2	19.8	14.7
18	0_0_1_1-1-all_0-0	19.8	16.7	11.2	12.5	15.1
19	0_1_1_0-0-all_0-0	15.5	14.0	11.2	18.5	14.8
20	0_1_1_1-0-all_0-0	17.7	15.7	9.5	16.5	14.9
21	0_1_0_1-1-all_0-0	12.0	18.2	11.2	18.7	15.0
22	0_1_0_0-0-all_0-0	15.3	14.2	11.2	20.2	15.2
23	0_1_1_1-1-all_0-0	16.7	15.7	11.2	17.8	15.4

overall best configuration (at position 1) does even not provide the best result in any of the test cases. There were only three configurations which were able to detect the error in the Crawler test case, and our heuristics (at position 2) was the best among them.

Overall, the table shows similar winners as in the comparison using total values presented in Table 4.12—the top positions are occupied by configurations which focus the noise on a single variable (having *one* within the configuration string), but our newly proposed heuristics (at positions 2 and 5) and the IBM ConTest random (at position 4) are still very well ranked. Our heuristics without the newly proposed noise strength computation achieved much better results than in the comparison using total

Table 4.16: A comparison of heuristics using *total/time* of Avio* results (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuraiton	Sunb.	Airl.	Craw.	FTPS.	TIDO.	aver.
1	0_1_1_1-0-one_0-0	4.8	7.0	**2.7**	9.2	7.7	6.3
2	0_1_0_1-0-one_0-0	4.5	5.0	**2.7**	11.5	8.8	6.5
3	0_0_0_0-0-all_1-0	7.6	**2.6**	17.7	**2.5**	**2.3**	6.5
4	1_0_0_0-0-all_0-0	5.3	10.3	3.2	10.3	5.0	6.8
5	0_1_0_1-1-one_0-0	4.0	11.8	5.0	11.8	8.7	8.3
6	0_1_1_1-1-one_0-0	**3.5**	9.5	7.5	15.7	7.2	8.7
7	0_0_1_1-0-one_0-0	9.7	5.2	18.0	3.0	9.3	9.0
8	0_0_0_1-0-one_0-0	10.0	6.3	17.8	2.8	8.7	9.1
9	0_0_0_1-1-one_0-0	7.8	8.7	15.8	10.8	7.0	10.0
10	0_0_1_1-1-one_0-0	10.5	10.8	14.5	9.5	7.5	10.6
11	0_0_0_0-0-all_0-0	12.0	5.0	19.2	11.0	10.5	11.5
12	0_0_0_0-0-all_1-1	6.7	3.7	23.0	19.0	13.0	13.1
13	0_1_0_1-0-all_0-0	15.3	14.7	5.5	19.0	14.7	13.8
14	0_1_0_0-0-all_0-0	12.3	17.2	7.5	17.0	16.0	14.0
15	0_1_0_1-1-all_0-0	11.0	17.3	6.8	18.2	17.2	14.1
16	0_1_1_1-0-all_0-0	17.3	17.8	6.5	13.7	15.5	14.2
17	0_1_1_1-1-all_0-0	16.8	14.3	9.3	16.0	14.7	14.2
18	0_0_1_1-0-all_0-0	19.8	13.0	14.3	9.5	16.7	14.7
19	0_0_0_1-1-all_0-0	16.0	14.0	16.8	14.3	13.7	15.0
20	0_1_1_0-0-all_0-0	14.8	16.2	10.3	17.8	16.5	15.1
21	0_0_1_1-1-all_0-0	19.8	15.3	16.0	9.7	15.2	15.2
22	0_0_0_1-0-all_0-0	16.3	18.2	17.3	11.8	17.3	16.2
23	0_0_1_0-0-all_0-0	19.3	19.2	18.5	9.8	18.0	17.0

values. This is mainly because the heuristics puts less noise into the execution still providing a good coverage. The configuration which uses random noise placement heuristics only is at position 11.

Table 4.16 shows results for the Avio* coverage. Our heuristics (at position 3) achieved the best results in three out of five test cases (Airlines, FTPServer, and TIDOrbJ). The heuristics was not evaluated as the overall winner due to the poor results that it achieved in the Crawler test case. Our heuristics with the newly proposed noise strength computation ended at position 12. This is caused by the proposed noise strength computation that sometimes puts a considerable amount of noise to places where it might be interesting. This leads to poor results in comparisons where the time plays an important role. On the other hand, the heuristics provided better results

Table 4.17: A comparison of heuristics using *total/time* of HBPair* results (value = average ordering (1=best), noiseFreq=150, noiseType=all, SeqLen=50)

Pos.	Configuration	Sunb.	Airl.	Craw.	FTPS.	TIDO.	aver.
1	0_1_1_1-0-one_0-0	5.8	7.3	**2.3**	8.0	**4.8**	5.6
2	0_1_0_1-0-one_0-0	4.3	4.8	**2.3**	12.2	6.3	6.0
3	0_0_0_0-0-all_0-1	4.8	**1.6**	16.2	4.2	7.0	6.8
4	1_0_0_0-0-all_0-0	4.5	10.0	5.2	9.5	5.0	6.8
5	0_1_0_1-1-one_0-0	4.8	11.7	4.7	11.0	7.0	7.8
6	0_0_0_1-0-one_0-0	11.3	6.2	15.8	2.8	5.0	8.2
7	0_1_1_1-1-one_0-0	4.5	9.5	6.5	14.8	7.0	8.5
8	0_0_1_1-0-one_0-0	11.0	5.3	16.0	**3.7**	7.3	8.7
9	0_0_0_0-0-all_0-0	**1.0**	4.8	17.3	13.5	12.8	9.9
10	0_0_1_1-1-one_0-0	13.2	11.8	15.3	9.3	5.5	11.0
11	0_0_0_1-1-one_0-0	12.0	9.3	16.5	10.3	8.7	11.4
12	0_0_0_0-0-all_1-1	6.0	3.0	23.0	19.7	13.0	12.9
13	0_1_0_1-1-all_0-0	11.2	17.0	7.8	17.3	15.8	13.8
14	0_1_0_1-0-all_0-0	13.3	14.3	6.2	20.7	14.8	13.9
15	0_1_1_1-0-all_0-0	16.2	17.3	6.0	13.2	17.5	14.0
16	0_1_0_0-0-all_0-0	10.8	17.2	8.2	18.0	16.7	14.2
17	0_1_1_1-1-all_0-0	15.0	14.3	9.8	14.7	18.5	14.5
18	0_0_0_1-1-all_0-0	15.7	13.3	18.0	13.3	13.5	14.8
19	0_0_1_1-0-all_0-0	18.8	13.5	14.8	11.0	15.7	14.8
20	0_0_1_1-1-all_0-0	20.7	15.0	16.7	8.3	15.8	15.3
21	0_1_1_0-0-all_0-0	14.2	16.7	10.2	15.7	20.2	15.4
22	0_0_0_1-0-all_0-0	13.8	18.7	18.0	14.0	13.8	15.7
23	0_0_1_0-0-all_0-0	20.8	19.8	19.2	9.2	19.2	17.6

than using our noise injection heuristics without the newly proposed noise strength computation in the comparisons using total numbers because it was able to examine more different interleavings.

The overall best results were obtained by the combination of the heuristics which focuses on a selected shared variable and the *timeoutTampering* and *haltOneThread* noise seeding heuristics (position 1). This is again mainly due to the effect of the *timeoutTampering* heuristics in the Crawler test case. Quite good results were obtained using the ConTest random setting (at position 4) again. The configuration which disables noise ended at position 11.

Finally, Table 4.17 shows results for the HBPair* coverage. There is no heuristics configuration providing the best result for more than two test cases in the table. Our heuristics (at position 3) achieved the best result in the Airlines test case. Moreover, our heuristics (at position 12) achieved the second best result in this test case. Our heuristics at position 12 achieved the absolutely worst rank in the Crawler test case mainly due to the high time degradation it introduced. This was mainly caused by a few accesses to shared variables among which the timeouts simulating the environment take place. Therefore, our noise strength computation algorithm inferred a high *max* value which led to a high amount of noise during the next executions of the test.

As in the previous tables, the overall good results are provided by the heuristics which focus noise on a single variable and also the IBM ConTest random setting (at position 4). The configuration which uses only random noise placement heuristics (at position 9) achieved the best rank in the Sunbank test case. This test case is rather primitive, and all configurations achieved the full coverage in it. Therefore, the Sunbank column shows ranks according to the overhead the configuration introduced. The configurations which focus noise only on a single variable introduced a smaller amount of noise than those which put noise everywhere. Surprisingly, the ConTest random option (at position 4) achieved a very good rank. This is probably because the option also sets the noise frequency parameter which seems to be on average lower than the value 150 which is used for other configurations. We are glad to see that both versions of our newly proposed heuristics (at positions 3 and 12) achieved good ranks in this column.

4.3.4 The Best Improvement Achieved by Noise-based Testing

The comparison of noise seeding techniques presented above indicates that using the random noise placement heuristics provides poor improvement of the testing process in some cases. Next, we presented a comparison of different noise placement heuristics which show that indeed the random noise placement heuristics only rarely achieve the best improvement. In this subsection, we focus on the best improvement achieved by noise-based testing in our experiments.

Table 4.18 presents the best relative improvement (denoted as *impr.* in the table) achieved in our experiments for all the considered metrics and test cases. The improvement is computed as relative improvement comparing to the configuration without noise injection. The next three columns (denoted as *nFreq*, *nType*, and *Configuration*) presents the used noise frequency, noise type, and a configuration encoding enabled advanced noise seeding techniques and noise placement heuristics. The meaning of the string used in the configuration column is the same as in the previous subsection. The table shows results for SeqLen=50. The improvement for the HBPair* metric and the Sunbank test case is not present because even the configuration without noise achieved the full coverage. The improvement for the error manifestation ratio (denoted as *Error*) and the TIDOorbJ test case is not present because the test case contains no error. The ★ symbol in the error manifestation ratio and the Crawler test case means that the improvement cannot be computed because

Table 4.18: The best improvement achieved by noise injection techniques (SeqLen=50)

Test	Metric	Impr.	nFreq.	nType	Configuration
Sunbank	Error	14.30	200	sleep	0_1_0_1-0-one_0-0
	Avio	1.62	200	bWait	0_1_1_1-0-one_0-0
	HBPair	–			
Airlines	Error	5.93	150	yield	0_1_0_1-0-one_0-0
	Avio	1.99	50	yield	0_0_0_0-0-all_0-0
	HBPair	1.90	50	yield	0_0_0_0-0-all_0-0
Crawler	Error	★	–	bWait	0_0_0_0-0-all_0-1
	Avio	8.20	50	mixed	0_1_1_1-0-all_0-0
	HBPair	3.55	200	mixed	0_1_1_1-0-all_0-0
FTPServer	Error	1.09	50	sleep	0_0_0_1-0-one_0-0
	Avio	1.26	50	wait	0_1_1_1-0-all_0-0
	HBPair	1.55	150	bWait	0_0_1_1-0-all_0-0
TIDOrbJ	Error	–			
	Avio	1.12	200	bWait	0_1_1_1-1-one_0-0
	HBPair	1.23	200	bWait	0_1_1_1-1-one_0-0

in our experiments the error does not manifest when the noise was disabled. The best value achieved by our heuristics reached 2 % of error manifestation in this test case (in average 1 error manifestation per 50 executions). If we compare this value with the value 0.06 % presented in Table 3.2, the improvement can be considered as quite high.

The improvement is high reaching several hundreds percents in some cases. The lowest improvement was achieved in the error manifestation ratio and the FTPServer test case. This is mainly because the error manifestation ratio is quite high even without the noise injection and fact that any performance degradation in effect makes the code containing the error execute less often. Overall, very good improvement has been achieved in the smaller test cases and the Crawler test (mainly due to *timeoutTampering* seeding technique). The improvement around 55 % and 22 %, respectively, in the HBPair* metric and the FTPServer and TIDOorbJ test cases also demonstrates the positive effect of relatively cheap and easy to use noise injection technique in the process of testing concurrent programs. Again, one cannot claim a clear winner among the noise seeding and placement heuristics.

4.4 Suggestions for Noise-based Testing

The results presented above indicate that there is no optimal configuration, and for each test case and each testing goal, a different setting of noise heuristics provide the best result. Moreover, using a wrong noise injection technique can in some cases degrade the quality of the testing process. Therefore, if no information concerning the tested program is available, a good option is to start with the IBM ConTest default configuration which has the IBM ConTest random parameter enabled. This parameter makes IBM ConTest select noise heuristics and their parameters at random before each execution. This setting does often not achieve the overall best results as shown above but it provides reasonably good results with a minimal effort.

Otherwise, one has to set up the noise seeding and placement heuristics manually. As for noise seeding heuristics, good results were often provided by the *yield*, *synchYield*, *wait*, and *busyWait* heuristics. The *yield* and *synchYield* heuristics have a minimal impact on the performance of the test as can be seen in Table 4.4 while still providing the best improvement in some cases. The *wait* and *busyWait* heuristics cause a considerable performance degradation but they can help to test even rarely executed synchronisation scenarios in complex programs as can be seen in Table 4.3. The presented results indicate that in most cases higher noise frequency does not mean a higher probability of spotting an error or higher coverage. On the other hand, a high noise frequency used with a demanding heuristics (e.g., *busyWait*) has a negative impact on the performance of the test as can be seen in Table 4.4.

Both the considered advanced noise seeding heuristics provide in some cases a considerable improvement of the testing process. Therefore, it is worth to enable them and test whether they positively affect results of the considered test case. Our results presented in Table 4.11 indicate that the performance degradation caused by these techniques is not high. Further, the impact of the *timeoutTampering* heuristics on tests which contain calls to timed `sleep` and `wait` methods is high. We therefore suggest to perform a simple static analysis which detects calls of these methods in the tested program and enables the *timeoutTampering* heuristics if such calls are present in the code.

As for noise placement heuristics, the heuristics which focus the noise on a single randomly chosen variable combined with the advanced noise seeding techniques and our newly proposed heuristics often provide the best results. We therefore suggest to prefer these heuristics which put noise only on carefully selected places to heuristics which simply put noise randomly or to too many places. If the performance degradation is not an issue, our heuristics with noise strength computation often provides better results than the heuristics without this feature. And, if the performance is important, our heuristics without the noise strength computation often provide the best results.

To sum up, although we provided same hints on using the noise techniques above, these advises are not definite since different testing scenarios can quite significantly vary as we proved by our experiments. Hence, if it is possible, we suggest to experiment with more noise settings. Moreover, in the next chapter, we propose an automated approach, based on using search techniques, for finding suitable noise settings.

4.5 Summary

In this chapter, a comparison of multiple noise injection heuristics that was missing in the current literature has been provided. We have also proposed a new, original noise injection heuristics, winning over the existing ones in some cases. It was shown that there is no silver bullet among the existing noise seeding or placement heuristics although some of them are on average winning in certain testing scenarios. Finally, several suggestions on how to test programs using noise injection technique has been proposed.

Chapter 5

Search-based Testing of Concurrent Programs

Search-based testing applies metaheuristic search techniques introduced in Chapter 1 to the problem of software testing. In order to apply metaheuristics to software engineering problems like testing, one has to consider the following steps [27]: (i) Decide whether the problem is suitable for search-based techniques, (ii) formulate the problem as a search/optimisation problem and define a representation for the possible solutions, (iii) define the fitness function, (iv) start with the simple Hill-climbing algorithm—if the results are encouraging, i.e., better than random search, consider other local search and genetic approaches, and (v) select an appropriate metaheuristic technique, its parameters and operators if necessary.

Many existing works summarised in the recently published surveys [81, 48, 3] show that various problems of software testing are indeed suitable for application of metaheuristics. In this chapter, the steps introduced above are followed to demonstrate that metaheuristic search techniques can be successfully used for testing of concurrent programs using the noise injection techniques.

In Chapter 4, absence of silver bullet among the many existing noise injection heuristics is shown. Results provided by them depend on the tested program and testing goal as well as on the run-time environment (the type and number of processors and the actual workload are usually the most significant factors). Actually, some configurations can decrease the probability of an error manifestation. This is helpful for run-time healing of errors [58], but it is highly undesirable for detecting them. Moreover, the number of possible settings of the noise injection (and also of the test itself) together with the considerable time needed to run a test in order to evaluate the efficiency of a certain noise configuration makes exhaustive searching for suitable noise configurations impractical. This is exactly the case where metaheuristic search techniques can help.

In this chapter, a new application of metaheuristic algorithms to search for suitable noise heuristics and their parameters is proposed. We formalise this task as the *test and noise configuration search problem* (the TNCS problem). Then, it is shown that even the simple hill-climbing algorithm can in some cases provide better results than the random search algorithm. Motivated by this success, a way how to represent instances of the TNCS problem for genetic algorithms is proposed. The way how to set parameters of genetic algorithms when solving the TNCS problem is also discussed. Next, the framework is instantiated by a concrete combined objective function suitable especially (but, as the experiments show, not only) for data race detection. Finally, the proposed approach is evaluated on a set of benchmarks, showing that it provides significantly better results than the so far preferred random noise injection.

The rest of the chapter is organised as follows. First, related work and previous attempts to apply metaheuristics to the problem of concurrent programs testing are discussed. Then, the testing problem as an optimisation problem is formulated and several suitable objective functions are discussed in Section 5.2. Next, results of our initial experiments with several versions of the hill-climbing algorithm on a few test cases is presented in Section 5.3. Then, in Section 5.4, the way how to utilise genetic algorithms to solve the TNCS problem is proposed. The problem of finding suitable values of parameters of genetic algorithm is discussed and experimentally evaluated in Section 5.5 and Appendix A. Next, our approach of applying a genetic algorithm to the testing of concurrent programs including a suitable incorporation of a run-time verification algorithm is introduced in Section 5.6. Finally, experimental evidence on how the proposed approach can improve the testing process is provided in Section 5.7.

5.1 Related Work

Most existing works in the area of search-based testing of concurrent programs focus on applying various metaheuristic techniques to control the state space exploration within the *guided model checking* approach [45]. Here, the intention is to explore areas of the state space that are more likely to contain concurrency errors even when the entire state space will not be explored. Hence, the testing problem is translated into the problem of *searching for a walk in a directed graph* [25]. The graph represents the state space generated by a model checker, and the walk starts in an initial state and ends in an objective node (an error state).

Within the guided model checking approach, various metaheuristic algorithms including simulated annealing [25], genetic algorithms [45, 8], the partial swarm optimisation (PSO) [25], and the ant colony optimisation (ACO) [6, 7] have successfully been applied to find deadlocks and/or assertion violations in models of rather simple concurrent programs and protocols. A candidate solution representing a path in the graph is often encoded as a vector of values each representing a transition the model checker should take in a particular step. This allows to search for solutions with a minimal path length but also brings a considerable memory overhead when a large model with very long paths is considered.

The fitness functions used in the above mentioned works are usually based on characteristics of the graph encoding the state space. For instance, in [45], the authors sum the number of enabled transitions from all states along a candidate solution path. This is motivated by the assumption that the number of enabled transitions is decreasing along the path, finally reaching zero when a deadlock is detected. In [6, 25], a fitness function prefering shorter paths (in terms of the number of transitions) with a high number of blocked threads and rewarding paths leading to a deadlock is used.

In [7], the ACO algorithm [104] is used to find liveness property violations encoded using a Büchi automaton [12]. Similarly to approaches above, the goal is to discover a path in the transition system from the initial state to a state where the considered property is violated. This time, the process is iterative. The ants look for a good path in the bounded neighbourhood of an initial state computed as the ending point of the best path discovered in the previous iteration (or initial state of the model). The fitness function is quite complicated but in general the ants look for accepting states of the Büchi automaton and cycles containing such states.

All mentioned works and a comparison presented in [25] report better results obtained by the newly proposed metaheuristic approaches in comparison to traditional model checking search techniques (mainly the random, breadth first search (BFS), and depth first search (DFS) algorithms [12]). Experimental evaluation is usually done on complex models representing communication protocols or simple concurrent programs.

An advantage of the guided model checking approach is that the underlying model checking offers a well-defined state space and a high degree of systematicity. On the other hand, the approach shares limitations of model checking in terms of scalability and cost of the environment modelling. In our approach, we focus on testing which is able to handle much larger real programs but does not provide such precision as the model checking approach.

In [34], an application of metaheuristics to the problem of debugging concurrent programs using noise injection is presented. Within this approach, the debugging problem is translated into the *test data generation* problem [81]. In this problem, the goal is to automatically select inputs of the test such that a chosen testing goal is achieved.

In particular, the work [34] uses a genetic algorithm to choose places in the code to inject noise in order to make a known error show up during the execution. The approach searches in the set of possible noise configurations C defined as the powerset of the disjoint union of sets S_V, S_A, and S_L where S_V defines the noise applied to selected program variables, S_A defines the noise applied to selected accesses to shared memory, and S_L defines the noise applied to selected concurrency related events (lock operations, etc.). The noise is determined by the type of noise together with the strength of the noise.

Each candidate solution is represented as a three level hash table. The first level contains the type of noise, the second level program locations identified by the heuristics, and the third level contains the noise strength for the particular noise type and location. The paper defines genetic algorithm operators as follows. Recombination is done by randomly choosing a pair of parent configurations each contributing by a random subset of its heuristics. Mutation is preceded by removing a subset of variables and locations from selected configurations. This is motivated by the intention to create smaller configurations that still manifest the concurrent bugs. The traditional role of the mutation operator, i.e., to introduce a new genetic material, is achieved by randomly creating a number of new configurations.

The paper also presents two objective functions ($size$ and $entropy$) and a fitness function computed as a weighted combination of the objective functions. The $size$ function maps each configuration $c \in C$ to a positive number representing the amount of noise produced by the configuration. The $entropy$ function maps each configuration c to a value in $[0,1]$ representing the probability of an error manifestation when the configuration is used. The fitness function therefore prefers configurations which make the error manifest with a high probability using a minimal amount of noise.

The technique is evaluated on a set of small (hundreds of lines) Java programs that contain known concurrency errors which manifest quite often when noise injection is used. Author claim that their approach is able to minimise the number of locations where to put noise and to increase the probability of an error manifestation. However, statistical data supporting this claim are missing in the paper.

Compared to [34], we do not search for concrete locations which should be noised with particular noise. Instead, we search for noise seeding and noise placement heuristics (or combinations of these heuristics) and their parameters which can provide good results for a particular test and environment. This allows us to use a simpler representation of individuals and to support much larger test cases with plenty of possible locations to be noised. Moreover, we propose new fitness functions which allow us to focus not only on debugging but also on testing. Further, we suggest a simple incorporation of various dynamic analyses into the approach. We also consider and reflect the non-deterministic behaviour of concurrent software. In particular, we evaluate each individual by a set of experiments and consider reevaluation of already evaluated individuals. Finally, we present our results on a small set of real concurrent programs of different size.

The same debugging problem of increasing the probability of a previously detected error manifestation is targeted in [14, 108] too, which, however, do not consider metaheuristic search algorithms. In [14], program locations are first statically classified according to their suitability for noise injection. Then, a probabilistic aloogorithm is used to find a subset of program locations that increase the error manifestation ratio. In [108], a machine learning feature selection algorithm is used to identify a subset of program locations where to inject noise. In this case, the test is executed many times, and program locations where the noise was injected in each execution are collected together with information whether the error has manifested. The algorithm then correlates program locations from executions where the error manifests.

5.2 Concurrent Programs Testing as a Search Problem

In this section, our proposal of how search techniques can be combined with noise-based testing of concurrent programs by identifying suitable combinations of noise injection heuristics as well as their parameters is presented. In particular, the proposed use of search techniques via the so-called test and noise configuration search (TNCS) problem is formulated. Subsequently, several objective functions that can be useful when dealing with various instances of the TNCS problem are discussed. These functions can typically act as building blocks of more complex combined objective functions as it is illustrated in Section 5.7.

5.2.1 The Test and Noise Configuration Search Problem

As already mentioned in the introduction, there are two main issues that must be solved when using noise injection. First, one needs to determine program locations where to insert noise. Heuristics which target this problem are called *noise placement heuristics*. Second, one needs to determine which *noise seeding heuristics*, i.e., which way of disturbing thread scheduling, should be used. Moreover, most types of the heuristics are adjustable by one or more parameters influencing their behaviour and efficiency (e.g., noise seeding heuristics are often parameterized by their strength). Further, one can combine several noise placement and seeding techniques within one execution. Indeed, our results presented in Chapter 4 show that such a combination provides in many cases better results than using a single heuristics. Finally, it is usually the case that there exist multiple test cases for a given program that can also be parametric.

With respect to the above, we formulate the *test and noise configuration search problem* (the TNCS problem) as the problem of selecting test cases and their parameters together with types and parameters of noise placement and noise seeding heuristics that are suitable for a certain test objective.

Formally, let $Type_P$ be a set of available types of noise placement heuristics each of which we assume to be parameterized by a vector of parameters. Let $Param_P$ be a set of all possible vectors of parameters. Further, let $P \subseteq Type_P \times Param_P$ be a set of all allowed combinations of types of noise placement heuristics and their parameters. Similarly, we can introduce sets $Type_S$, $Param_S$, and S for noise seeding heuristics. Next, let $C \subseteq 2^{P \times S}$ contain all the sets of noise placement and noise seeding heuristics that have the property that they can be used together within a single test run. We denote elements of C as *noise configurations*. Further, like for the noise placement and noise seeding heuristics, let $Type_T$ be a set of test cases, $Param_T$ a set of vectors of their parameters, and $T \subseteq Type_T \times Param_T$ a set of all allowed combinations of test cases and their parameters. We let $TC = T \times C$ be the set of *test configurations*.

Now, the TNCS problem can be expressed as searching for a test configuration from TC suitable wrt. some given objective function. One can also consider the natural generalisation of the TNCS problem to searching for a set of test configurations, i.e., a member of 2^{TC}, suitable wrt. some given objective function.

5.2.2 Objective Functions for the Context of the TNCS Problem

Next, several possible objective functions that can be useful in various instances of the TNCS problem are presented. They are typically combined into more complex objective functions as it is illustrated in Section 5.7.

First, an objective function that can often be found useful is to minimise the impact of noise injection on the *time of execution* of a test case. The more noise is injected into the execution the slower the execution typically is. The slowdown can be unwelcome especially when the time for testing is limited. Then, due to the slowdown, less executions of a test case and/or less test cases will be considered which may in turn negate the aim of using noise injection to improve the quality of testing. The time aspect is also important when a program under test needs to meet certain throughput or response time requirements that could be broken by an excessive use of noise.

Next, since the primary goal of testing is to find errors, a natural objective function is to maximise the *number of errors* that occur (and are detected by the test harness) when executing tests with a certain configuration. Once some test configuration is found suitable wrt. the number of errors it allows one to observe, one could think that this configuration is not useful any more since the errors were already detected. However, this test configuration can be used for further testing in hope that it will allow one to discover even more errors (recall that due to the non-determinism of scheduling, not all errors will show up in a single run or a set of runs). Moreover, one can also think of using this test configuration in regression testing or when testing similar applications.

Another sensible objective function, tightly related to the above, is to monitor test executions under particular test configurations by some *dynamic analyser* and to maximise the number of warnings about dangerous behaviour of the program under test that get reported. Test configurations delivering good results in this case can subsequently be used for more extensive testing in hope of finding a real error even though an actual error was not seen during evaluation of the test configuration. The reliability of this approach of course depends on the precision of the chosen analyser. A high ratio of false positives and/or negatives makes this objective function unreliable.

A further possibility is to use a suitable *coverage metric* allowing one to judge how much of the possible behaviour of the program under test has been covered (and hence how likely it is that some undesired behaviour was omitted) and to look for test configurations maximising the obtained coverage. Concurrency-related metrics based on dynamic analyses which we presented in Chapter 3 can be especially useful here. These metrics are not based on simply counting the number of produced warnings, but on much finer measures. Some of them are based on monitoring events that make the internal state of a dynamic analyser change, e.g., the *HBPair* metric based on the happens-before relations, and some express how many internal states a certain dynamic analyser reached, e.g., the *GolidLockSC* metric based on monitoring the internal states of the GoldiLock analyser [32]. Of course, there are many other existing coverage metrics which can be considered as mentioned in Chapter 3. For instance, the synchronisation coverage [22] (*Synchro*) which measures how well the various synchronisation mechanisms used in the program under test are tested (by measuring how many different scenarios of the use of the synchronisation mechanisms were witnessed). A drawback of many concurrency coverage metrics is that it is often impossible to compute what the full coverage is; this is, however, not a problem here since we are interested in relative comparisons of the coverage achieved through different test configurations.

Fitness of a test configuration $tc \in TC$ wrt. the above objective functions has typically to be evaluated by a *repeated execution* of the test case encoded in tc with the test parameters and noise configuration that are also a part of tc. Recall that the noise configuration can contain multiple types of noise heuristics. We assume all of them to be used in each testing run, which is consistent with our definition of noise configurations that allows for only those combinations of noise heuristics that can indeed be used together. Further note that the repeated execution makes sense due to the non-determinism of thread scheduling. The evaluation of individual test runs must of course be combined, which can be done, e.g., by computing the *average evaluation* or by computing a *cumulative evaluation* across all the performed executions.

In addition, it is also possible to define some simple objective functions directly on the test configurations. For instance, one can minimise/maximise the number of enabled heuristics, volume or frequency of noise to be injected, etc. Such objective functions are typically not sensible alone, but can make sense when combined with other objective functions. Fitness of a given test configuration wrt. such objective functions can be evaluated *statically*, i.e., without any test execution.

5.3 Initial Experiments with the Hill-Climbing Algorithm

In this section, our initial experiments done with the basic local search algorithm—the Hill-climbing algorithm [104] are presented. In the experiment, the Hill-climbing algorithm is compared with the random search approach for solving the simple TNCS problem. The search space is built using parameters of the IBM ConTest tool and one parameter which controls the number of threads (or clients) used within the test. In particular, two noise placement heuristics implemented in ConTest are considered: the *random* heuristics which picks program locations randomly and the *sharedVar* heuristics which focuses on accesses to shared variables. Both heuristics are described in Chapter 4. Default values of parameters of the *sharedVar* heuristics are considered only. Recall that both of the considered noise placement heuristics inject noise at places they select with a given probability. The probability is set globally by a *noiseFreq* setting from the range 0 (never) to 1000 (always). The values of the *noiseFreq* parameter are limited to 11 values (0, 100, ..., 900, 1000).

Further, 5 basic and 2 advanced noise seeding techniques which are described in Chapter 4 are considered. In particular, the following basic heuristics are used: *yield*, *sleep*, *wait*, *synchYield*, and *mixed*. The basic techniques cannot be combined, but any basic technique can be combined with one or both considered advanced techniques—the *haltOneThread* and the *timeoutTamper* techniques.

For the experiments, three concurrent programs containing concurrency errors are used, namely, Airlines, Crawler, and FTPServer which are described in Chapter 3. Each test case is adjusted by the *NumThreads* parameter which specifies how many competing threads should be executed (or how many client processes may connect to the FTPServer concurrently). The considered values of this parameter were 2, 4, 8, 16, 32 for the Airlines and Crawler test cases and 2, 4, 8, 16 for the FTPServer, respectively.

The search space of possible noise and test configurations for the experiments can be expressed using a set of vectors of numbers in the range $(0,0,0,0,0,0)$–$(11,1,4,1,1,4-5)$. Here, the first entry controls the *noiseFreq* setting, the next entry controls the *sharedVar* noise placement heuristics. The next three entries control the setting of the basic and advanced noise seeding heuristics. Finally, the last entry controls the *NumThreads* test parameter. The search space therefore contains only 2200 states in cases of the Airlines and Crawler or 1760 states in the case of the FTPServer test case only.

The Hill-climbing algorithm needs neighbourhood to be defined. We define the neighbourhood of each state as the set of states which differ from the particular state in one element of the vector only. The *noiseFreq* and *NumThreads* are considered to be ordinal. Therefore, only difference by one is considered to be neighbouring in the particular elements of the vector.

Our infrastructure for search-based testing SearchBestie which is described in Appendix B was used to evaluate each state of the state space 100 times collecting two concurrency-related coverage metrics: (i) *Synchro* which tracks whether each synchronisation mechanism does something useful and (ii) *ConcurPairs* which tracks context switches among concurrency related locations. Both metrics are described in Chapter 3. This way, a database of 220,000 (176,000) results for each test case was obtained. The SearchBestie than used these results (provided for each state in

Table 5.1: Results of searching for the best configuration

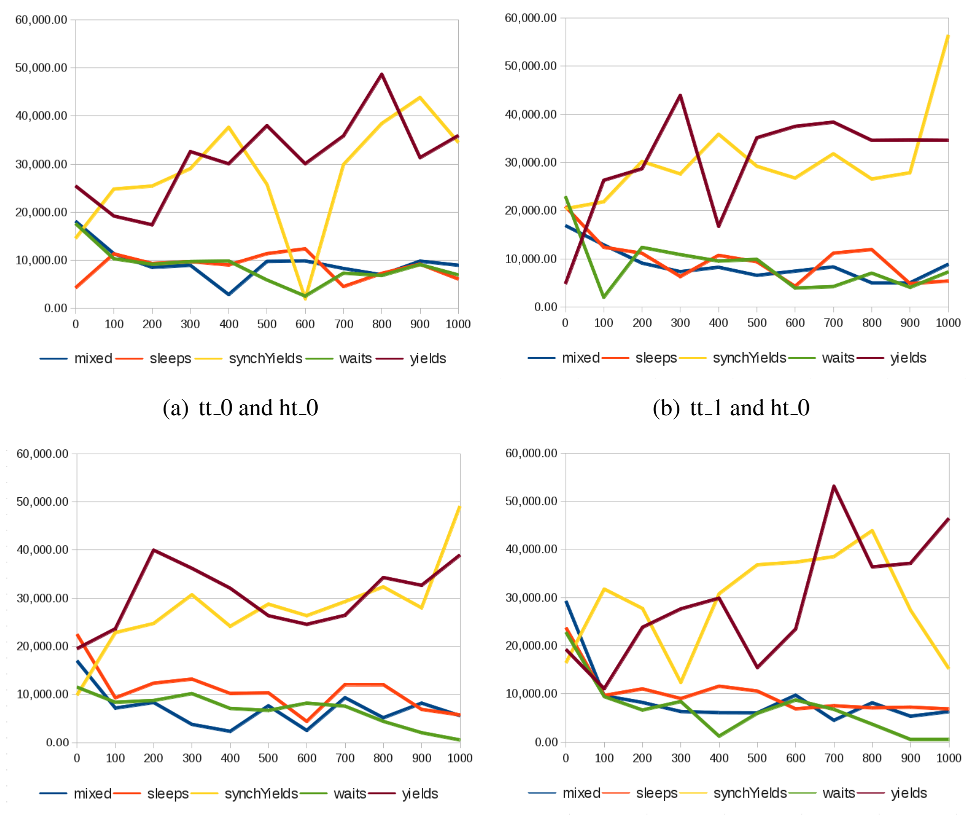
	ConcurP. comparison			Synchro comparison		
Test case	States	Hill	Rand	States	Hill	Rand
Airlines	344	25 %	17 %	326	100 %	100 %
WebCrawler	366	36 %	16 %	371	14 %	16 %
FTPServer	277	11 %	22 %	301	29 %	21 %

the state space in the same ordering) during each evaluation of the search algorithm instead of evaluating each noise configuration by executing the test. This allows us to compare the search algorithms using the same results without any influence of the non-determinism of scheduling, i.e., the SearchBestie provided the same fitness values for the same states in the state space. Experiments with the first two programs were performed on Intel Core 2 Duo E8400 with 2 GB RAM and experiments with the FTPServer on 2xIntel Xeon X5355 with 64 GB RAM. All machines ran 64-bit Linux and Java 6.

The goal was to search for a suitable configuration of ConTest that provides the best coverage. To achieve this goal, the fitness function was set to represent the accumulated number of tasks of the considered coverage metric covered in the particular state of the state space. The testing scenario was as follows: First, the maximal value that can be reached was identified (the best state in the state space). Then, the Hill-climbing algorithm was run 100 times and got the average number of steps that Hill-climbing needed to reach the optimum (global or local). After each exploration, it was checked whether the algorithm reached the global optimum. Finally, the random search algorithm was run 100 times too, and the number of times the algorithm found the global optimum was counted.

Table 5.1 summarises our results. Column *Test case* contains the name of the test. The rest of the table is divided into two parts. The first part denoted as *ConcurP. comparison* gives a comparison of the Hill-climbing and the random search algorithms when trying to maximise the *ConcurPairs* coverage metric. Similarly, the second part denoted as *Synchro comparison* compares the search algorithms when trying to maximise the *Synchro* coverage metric. Each comparison contains three columns. Column *States* shows how many states were explored on average by our Hill-climbing algorithm until the optimum was reached, and the two other columns show the percentage in which the algorithms found the global optimum. Results for both considered coverage measures are provided.

It can be seen that in 3 cases, the Hill-climbing algorithm beats the random approach, and in 2 cases, it does not. In the case of the *Airlines* example and the *Synchro* coverage metric, many states represent the global optimum and therefore both algorithms reached 100 %. The number of explored states in the case of *FTPServer* is lower because the state space itself is smaller (1760 states). Our results indicate that



(a) tt_0 and ht_0

(b) tt_1 and ht_0

(c) tt_0 and ht_1

(d) tt_1 and ht_1

Figure 5.1: Fragments of landscape for the FTPServer test case, 8 client processes, and the ConcurPairs metric (the horizontal axis gives the value of noiseFreq parameter, the vertical axis gives the cumulative number of covered tasks)

the metaheuristic algorithms can be used to solve the TNCS problem because even the simple Hill-climbing algorithm is in some cases able to overcome the random approach. But, the low success of the Hill-climbing algorithm in reaching the global optimum indicates that the method does not work well and often get stuck in a local optimum.

Therefore, we analysed the produced landscapes. Figure 5.1 illustrates our findings. The figure shows fragments of landscape generated for the FTPServer test case and the *ConcurPairs* coverage metric when *NumThreads* was set to 8 and *sharedVar* heuristics was disabled. The four subfigures captures landscape fragments for different configurations of the *timeoutTamper* (tt_) and *haltOneThread* (ht_) parameters which neighbour in our setting. The figures indicate that the landscape contains many local optima which are hard to overcome for the simple Hill-climbing algorithm. Similar situation was also observed for the other considered test cases and parameters. This observation corresponds to the results obtained when comparing different noise techniques which are presented in Chapter 4[1].

The analysis of landscapes produced by the *Synchro* coverage metric shows that these landscapes contain plateaus, i.e., areas with the same value of the fitness function, which prevent the Hill-climbing algorithm from reaching the global maximum. This was caused by the very limited number of different values provided by this metric. This observation corresponds to the results presented in Chapter 3.

5.4 A Genetic Approach to the TNCS Problem

In this section, we present our proposal of using a genetic approach to solve the TNCS problem. We again start by presenting the particular set of ConTest noise configurations considered. Test parameters are omitted in this section but are considered within experimental evaluation later on. Subsequently, we present how one can apply the genetic approach in this setting.

ConTest-based Noise Configurations

We consider noise injection heuristics implemented in ConTest extended by our plug-in implementing a coverage-based noise placement heuristics introduced in Chapter 4. Hence, we consider three noise placement heuristics: the *random* heuristics which picks program locations randomly, the *sharedVar* heuristics which focuses on accesses to shared variables, and our *coverage-based* heuristics which focuses on accesses near a previously detected thread context switch. The *sharedVar* heuristics has two parameters modifying its behaviour with 5 valid combinations of its values. The *coverage-based* heuristics is controlled by 2 parameters with 3 valid combinations of values. Recall that the *random* heuristics is enabled by default when $noiseFreq > 0$. But, in the considered setting, the *random* heuristics can be suppressed by one parameter of the *sharedVar* heuristics which explicitly disables the combination of these two heuristics. We consider the following 6 basic noise seeding techniques: *yield*, *sleep*, *wait*, *busyWait*, *synchYield*, and *mixed*. And both advanced noise seeding techniques: *haltOneThread* and *timeoutTamper*.

Individuals, Their Encoding, and Genetic Operations on Them

In order to utilise a genetic algorithm to solve the TNCS problem with the considered set of noise configurations, we let the particular test configurations play the role of *individuals*. We encode the test configurations as *vectors of integers*. The test configuration is either reduced to solely a noise configuration (when a single test case

[1]Results presented in this section were actually obtained before our work on concurrency coverage metrics presented in Chapter 3 and on noise techniques presented in Chapter 4. Therefore, these results partially motivate our further research presented in these chapters.

without parameters is considered), or it consists of the noise configuration extended by one or more specific entries controlling the test case settings. We, however, concentrate here on the noise configurations only, which form vectors of numbers in the range $(0,0,0,0,0,0)$–$(1000,5,3,6,2,2)$. Here, the first entry controls the *noise-Freq* setting, the next two control the *sharedVar* and *coverage-based* noise placement heuristics. The last three entries control the setting of the basic and advanced noise seeding heuristics. Each entry in the vector is annotated by a flag saying whether there exists an ordering on the values of the entry. We call entries whose values are ordered as *ordinal entries*.

We consider the standard one-point, two-point, and uniform element-wise (any-point) *crossover operators* [104] in the form they are implemented in the ECJ library [114]. *Mutation* is also done on an element-wise basis, and it handles ordinal and non-ordinal entries differently. Non-ordinal entries are set to a randomly chosen value from the particular range (including the current value). Ordinal entries (e.g., entries encoding the strength of noise or the parameter controlling the number of threads the test should use) are handled using the standard Gaussian mutation [104] (with the standard deviation set to 10 % of the possible range or minimal value 2). Finally, we consider standard proportional and tournament-based fitness selection operators [104] as they are implemented in the ECJ library.

5.5 Parameters of Genetic Algorithms and the TNCS Problem

Genetic algorithms are adjustable through a number of parameters influencing the efficiency of the search process. The way these parameters should be set to obtain a high efficiency usually depends on the considered problem. In this section, we provide our findings on how to set the parameters of genetic algorithms when solving the TNCS problem. We focus mainly on the following questions: How to set up the breeding infrastructure, i.e., which standard selection and crossover operators should be used, how to set up their parameters, which value of mutation probability provides good results, and whether elitism or random generation of individuals can help. We also target the question whether it is better to run a few big generations or instead more small generations in case the time for testing is limited.

Experiments for Finding Suitable Parameters of Genetic Algorithms

We conducted all our experiments aimed at finding a suitable setting of the parameters of genetic algorithms on one selected case study only. This is mostly due to the high time consumption of evaluating each test configuration through multiple test executions. In particular, we used the *Crawler* test case described in Chapter 3. We conducted our experiments on multiple machines, all having Intel i5 661 processors, running 64-bit Linux and Java 6. We used our infrastructure SearchBestie and IBM ConTest to evaluate test configurations and the ECJ library [114] to implement the genetic algorithms. We narrowed the search space down by sampling the *noiseFreq* parameter by ten, i.e., by reducing its possible values to 0, 10, ..., 1000.

With the aim of observing as many behaviours differing in their various important concurrency-related aspects as possible, we considered an objective function maximising the obtained coverage under three different concurrency coverage metrics introduced in Chapter 3, namely, *Synchro*, *Avio** and *HBPair**. This objective function covers three different aspects of concurrency behaviour: interleaving of accesses from different threads to shared memory locations via *Avio**, successful synchronisation of program threads inducing a happens-before relation via *HBPair**, and information about whether the implemented synchronisation does something helpful via *Synchro*. We used results of approximately 1 million randomly noised executions to estimate the 100 % achievable coverage (denoted as *max* below) for each of the metrics and set up the following fitness function:

$$\frac{1}{3} * \left(\frac{Avio^*}{Avio^*_{max}} + \frac{HBPair^*}{HBPair^*_{max}} + \frac{Synchro}{Synchro_{max}} \right)$$

The evaluation of each test configuration consisted of 5 executions of the test case with the noise parameters encoded in the test configuration. The value of the fitness function was then computed using the accumulated coverage of all the five executions.

We fixed the number of evaluated individuals to 2000 in each experiment. According to our experiments, this value is sufficient to reach saturation of the selected coverage metrics in the Crawler case study. We set the size of the considered populations and number of generations as follows (population/generation size): 200/10, 80/25, 40/50, 20/100, and 10/200. We considered the breeding infrastructure to consist of two selection operators which select individuals for the crossover operator. The output of the crossover operator was mutated using the mutation operator described in Section 5.4. We also let the best individual (one elite) to be added to the next generation without breeding.

We performed three sets of experiments. In the first one, we considered the standard fitness proportional selection operators, four different standard crossover operators (*one-point*, *two-point*, and *any-point* with the probability of mutating each element of the vector set to 0.1 and 0.25), and four different probabilities of applying the mutation operator (0.01, 0.1, 0.25, and 0.5). In the next two sets of experiments, we fixed the considered size of the population to 40, the crossover operator to *any-point* with probability 0.1, and the mutation probability to 0.01. We then studied the influence of elitism which puts into the next generation a number of individuals (0, 2, 4) without breeding, and a random creation of a few individuals (0, 2, 4) that are put into the following generation within the second set of experiments. In the third set of experiments, we focus on the influence of different selection operators on the obtained results. Within this set of experiments, we considered the fitness proportional and tournament selection operators (with the size of the tournament being 2 or 4) and some of their combinations.

From each experiment, we collected various data concerning the generated populations including, in particular, the following three statistics: (i) The average fitness value in each generation *aver* and (ii) the best individual fitness in each generation *best*, and (iii) the cumulative value of fitness from all already evaluated individuals *gcumul*. Our goal was then to identify parameters of the genetic algorithms under which the best test configuration out of all discovered test configurations is found, and it is found as quickly as possible. For that, we used the *best* and *aver* statistics. Moreover, we used the *gcumul* statistics to monitor cumulative value of the fitness along the breading process. The results of the experiments are summarised below with some more commented graphs available in Appendix A.

Results of Experiments with the Parameters of Genetic Algorithms

The values of the *best* and *aver* statistics that we obtained from the first set of experiments presented above show that small populations combined with the *any-point* crossover and mutation set to 0.01 are able to find the best individual (i.e., the best test configuration) out of all the encountered ones quite fast (within a few generations). Very small populations (10 and rarely also 20) are, however, sometimes not able to find the best individual and get stuck in a local optimum. On the other hand, in larger populations, it takes much longer to arrive to the best individual. The *any-point* crossover operator outperformed the other two operators, but one has to be careful about the probability used: the operator sometimes does not change the individuals when a low probability (0.1 or less) is used.

The best individuals obtained by the genetic algorithm in our experiments had fitness higher than 0.5, and they therefore covered more than 50 % of the concurrent behaviour as defined by our fitness function. The overall best individuals achieved fitness 0.64. The average fitness of the final population was in the worst case 0.35 only, which is quite similar to fitness 0.33 that we achieved by randomly generating individuals to evaluate. The highest average fitness was close to the maximum fitness of 0.64, which represents a situation when nearly all individuals in the generation were the same.

In the second set of experiments from the above, we clearly saw the positive effect of elitism (set to 10 % of the population). The selection operators seem to affect the results only a little. The best results seem to be provided by a combination of the tournament selection operator (with the size of the tournament set to a high value) and the fitness proportional selection operator.

Based on the results summarised above and presented in Appendix A, we found as suitable the following setting of the parameters of genetic algorithms for the considered concretisation of the TNCS problem: Size of population 20, two different selection operators (tournament among 4 individuals and fitness proportional), the *any-point* crossover with a higher probability (0.25), a low mutation probability (0.01), and two elites (that is 10 % of the population). We choose the low mutation probability 0.01 despite our results indicate that the individuals with highest fitness are most often found using the higher probability (0.25). This decision is motivated by our intention to prefer exploitation over exploration as explained below. This parameter setting is used in the experiments presented in the next section.

5.6 A Concrete Application of the Proposed Approach

In this section, we first propose a complex objective function for the TNCS problem that carefully combines the above discussed basic objective functions, finally leading to a concrete application of genetic algorithms for improving the process of testing of concurrent programs. In particular, the stress is on looking for data races, but as our experiments show, the approach helps in finding other kinds of concurrency-related errors too. Next, we present a collection of benchmarks and results of experiments with them which illustrate the efficiency of our approach.

A GoldiLocks-based Objective Function

Based on our experience with different concurrency coverage metrics and dynamic error detectors, we have decided to build our concrete objective function on maximising the coverage obtained under the concurrency coverage metric *GoldiLockSC* presented in Chapter 3 which is based on the GoldiLocks algorithm [32], together with maximising the number of actual warnings produced by this algorithm. We have chosen the GoldiLocks algorithm for our objective function because it has a low ratio of false positives, and it is able to continue in the analysis even after an error is detected. Moreover, our results indicate that the concurrency coverage metric *GoldiLockSC* has multiple positive properties. In particular, the coverage under this metric usually grows smoothly (i.e., with a minimum of shoulders) and does not stabilise too early (i.e., before most behaviours relevant from the point of view of data race detection are examined). Further, based on the discussion presented in Section 5.2.2, we also reflect in our objective function an intention to minimise the execution time and to maximise the number of detected errors.

In summary, we thus aim at (1) maximising coverage under the concurrency coverage metric *GoldiLockSC*, (2) maximising the number of warnings produced by the GoldiLocks algorithm, (3) maximising the number of detected real errors due to data races, and (4) minimising the execution time. The different basic objectives are combined using a system of weights assigned to them.

To be more precise, the *GoldiLockSC* metric counts the encountered internal states of the GoldiLocks algorithm (here, SC stands for the optimised version of the algorithm with the so-called *short circuits*, i.e., cheap checks done before the full algorithm is used). We weight the different coverage tasks of this metric as well as the error manifestation according to their severity. In particular, the coverage tasks of the *GoldiLockSC* metric are tuples $(ploc, state)$ where *ploc* identifies the program location at which some shared memory location is accessed, and $state \in \{SCT, SCL, LS, E\}$ denotes the internal state of the GoldiLocks algorithm. We divide the tasks into three categories according to severity of their *state*. The *SCT* state represents a situation where the first short circuit check of GoldiLocks (checking whether a variable is accessed by a single thread only) proves correctness of the given access. This situation is common for sequentially executed code, and so we assign it weight 1. The *SCL* and *LS* states mean that the first check does not succeed, but it is possible to use further heuristic short circuit checks (*SCL*) or use the full algorithm (*LS*) to infer a lock (or locks) whose locking proves correctness of the access. We assign such tasks with weight 5. Finally, the *E* state means that the algorithm detected a data race and produced a warning message. We weight such tasks with 10. We denote the weighted coverage as *WGoldiLockSC*.

A GoldiLocks warning has the form of a tuple $(var, ploc_1, ploc_2)$ where *var* identifies a shared variable, and $ploc_1$, $ploc_2$ represent two program locations between which a data race was detected. Sometimes, a single coverage task with $state = E$ produced at $ploc_1$ leads to several warnings differing in the $ploc_2$ or *var* values. We denote by *GLwarn* the number of different warnings issued during the test execution, and we give them the weight of 1000.

Finally, as we have already mentioned, we also aim at maximising the number of detected error manifestations (*error*) and minimising the execution time (*time*). Error manifestations are detected by looking for unhandled exceptions. They are given a very high weight of 10000. With respect to all the described objectives, we then define the fitness function as follows (expecting the time to be measured in milliseconds):

$$\frac{WGoldiLockSC + 1000 * GLwarn + 10000 * error}{time}$$

Case Studies

We concentrate primarily on data race detection, but we also try to apply our genetic approach to case studies containing other kinds of concurrency errors (and, as we will show, we obtain quite positive results even in such cases). In particular, we evaluate our approach on 5 test cases containing concurrency-related errors. The test cases are listed in Table 5.2. In the table, the *Param* column indicates the number of the test case parameters and the number of possible values of each parameter (e.g., $2,3$ means that the test takes two parameters, the first with two possible values and the second with three possible values).

The *Airlines*, *Crawler*, *Rover*, and *FTPServer* test cases were introduced in Chapter 3. The *Animator* test case which was also used as a benchmark in [15] is based on a simple graphic application for algorithm animation called *XtangoAnimator*. The program creates a window and draws a picture according to a given batch file. The test case contains data races which rarely lead to an unhandled exception. The test case consists of 31 classes and has 1.5 kLOC.

The *Animator*, *Crawler*, and *FTPServer* test cases contain a data race which leads to unhandled exceptions. The *Airlines* case study contains a high level atomicity violation that is detected by a final check at the end of the execution which throws an unhandled exception. Finally, the *Rover* test case contains a deadlock and an atomicity violation which leads to an unhandled exception.

We admit that the described case studies are not very large, and one could surely found much bigger ones. Let us, however, stress that the reason why we did not consider truly large benchmarks is *not* a bad scalability of our approach, but rather the large number of experiments that we did with the various parameter settings which in summary take a lot of time even on smaller benchmarks.

The *Airlines* and *Animator* test cases were run on Intel Core2 6600 machines, the *Rover* test case on a machine with an Intel i5-2500 processor, and the *FTPServer* test case on a machine with two Intel X5355 processors. In case of the *Crawler* test case, two different hardware environments were used. The first (denoted simply as *Crawler* in Table 5.2) used a machine with an Intel i5-661 processor, while the second (denoted *Crawler**) was executed on a machine with four AMD Opteron 8389 processors. These two options were used on purpose in order to study how our approach works in different hardware environments. All mentioned computers ran 64-bit Linux and Java version 6.

5.7 Experimental Results

To evaluate the efficiency of our approach when using the GoldiLocks-based objective function, we again used the infrastructure described in Section 5.4. We use the setting of parameters of genetic algorithms inferred in Section 5.5. Although this setting was inferred for a different objective function and using sampled values of the *noiseFreq* parameter only, we believe that it represents a good option even for other experiments with our genetic algorithm. Indeed, the objective function used in Section 5.5 was designed to be rather general in order to cover a lot of different concurrent behaviours. Moreover, we analysed the correlation between the values of the fitness function of Section 5.5 and the *GoldiLocksSC* metric used in the GoldiLocks-based objective function on the performed experiments and realized that the correlation is high. After all, the combination of *HBPair** and *Avio** focuses on the same events as the GoldiLocks algorithm.

Table 5.2: An experimental comparison of the proposed genetic approach with the random approach to setting test and noise parameters

Test case		Best configuration			Search process		
Name	Params	Gen.	Error	Time	Error	Error*	Time
Airlines	5,5,10	15	3.0 / 1.7	3.8 / 2.5	3.2	8.8	3.0
Animator	–	25	21.8 / 10.9	1.1 / 1.3	4.3	5.4	1.3
Crawler	–	22	– / –	1.3 / 1.5	0.3	1.1	3.3
Crawler*	–	25	– / –	1.1 / 1.1	0.4	1.0	2.8
FTPServer	10	14	1.2 / 1.0	3.8 / 4.7	0.9	1.7	1.9
Rover	7	3	★	33.7 / 19.4	3.2	8.8	3.0

In the experiments, we allowed the elite individuals to be re-evaluated in the following generations. This is motivated by the fact that a few executions of an individual (5 in our case) are often not sufficient to prove whether the configuration can make a concurrency error manifest. Indeed, tricky concurrency-related errors manifest very rarely even if a suitable noise heuristics is used as shown in Chapter 4. The reevaluation of elites therefore gives the most promising individuals another chance to spot an error. This setting is a compromise between a high number of executions needed to evaluate every individual more times and the available time we have.

We compare our genetic approach with the random approach to the choice of noise heuristics and their parameters. In the random approach, we randomly select 2000 test and noise configurations and let our infrastructure evaluate them in the same way we evaluate individuals in the genetic approach. Table 5.2 summarises our results. The table is based on average results obtained from 10 executions of the genetic and random approach. It is divided into three parts. In the left part (*Test case*), the test cases are identified, and their size and information about their parameters are provided.

5.7.1 An Evaluation of the Best Individuals

The middle part of Table 5.2 (*Best configuration*) contains three columns which compare the best individual obtained by our genetic approach and found by the random approach. The *Gen.* column contains the average number of generations (denoted as *gen* below) within which we discovered the best individual according to the considered fitness function. The numbers indicate that we are able to find the best individual according to the considered fitness function within the first quarter of the considered generations. This motivates our future work to design a suitable termination condition for our specific testing process.

The *Error* column of the *Best configuration* section of Table 5.2 compares the ability of the best individual to detect an error. The column contains two values (x_1/y_1). The first value x_1 is computed as the fraction of the average number of errors found by the best individual computed by the genetic algorithm and the average number of errors discovered by the best individual found by the random generation provided that an equivalent number of executions is provided to the random approach (this number is computed as *gen* times the size of the population which is 20). The second number y_1 is computed as the fraction of the average number of errors found by the best individual computed by the genetic algorithm and the average number of errors discovered by the best individual found randomly in 2000 evaluations. The –/– value represents a situation where none of the best individuals was able to detect the error within the allowed 5 executions. The $\bigstar$ symbol means that the genetically obtained best individual did not spot any error while the best individual found by the random generation did (we discuss this situation in more detail below).

Similarly, the *Time* column of the *Best individual* section of Table 5.2 compares average times needed to evaluate the best individual obtained by our approach and the best individual found by the random approach. Again, two values are presented (x_2/y_2). The first value x_2 is computed as the average time needed by the best individual found by the random approach if only $gen * 20$ evaluations are considered, divided by the average time the genetically found best individual needed. The second value y_2 shows the average time needed by the best individual found by the random generation when it was provided with 2000 evaluations, divided by the average time needed by the genetically found best individual.

The values that are higher than 1 in the *Error* and *Time* columns of the *Best individual* section of Table 5.2 represent how many times our approach outperforms the random approach. In general, one can see that the best individual found by our genetic approach has a higher probability to spot a concurrency error, and it also need less time to do so. Even if we let the random approach to perform 2000 evaluations, our best individual is still better. Exceptions to this are the *Rover* and *Crawler* test cases. In the *Crawler* test case, the error manifests with a very low probability. The best individuals in both cases were not successful in spotting the error (note, however, that the error was discovered during the search process as discussed below). In the *Rover* test case, the best individual found by the genetic algorithm was not able to detect an error and some of the best individuals found by the random approach did detect the error (as again discussed below, the error was discovered during the search process too). This results from the fact that the genetic approach converged to an individual that allows a very fast evaluation (over 30 times faster than the best configuration found by the random generation). This, however, lowered the quality of the found configuration from the point of view of error detection, indicating that as a part of our future research, we may think of further adjusting the fitness function such that this phenomenon is suppressed.

5.7.2 An Evaluation of the Search Process

The right part of Table 5.2 (*Search process*) provides a different point of view on our results. In this case, we are not interested in just one best individual learned genetically or by random generation that is assumed to be subsequently used in debugging or regression testing. Instead, we focus on the results obtained during the search process itself. The genetic algorithm is hence considered here to play a role of heuristics that directly controls which test and noise configurations should be used during a testing process with a limited number of evaluations that can be done (2000 in our case).

This part of the table contains three columns which compare the genetic and random approaches wrt. their successes in finding errors and wrt. the time needed to perform the 2000 evaluations. The first column (*Error*) compares the average number of errors spot during the search process and the average number of errors spot during the evaluation of 2000 randomly chosen configurations of the test and noise heuristics. The *Error** column compares the average number of errors detected by our genetic approach with the average number of errors spot by the random approach when the random approach is provided with the same amount of time as the genetic approach. Finally, the *Time* column compares the average total time needed by the random approach in 2000 evaluations and the average time needed by our genetic approach. Again, the values higher than 1 in all the columns represents how many times our approach outperforms the random approach.

The cumulative results presented in the *Error* and *Error** columns show that our approach mostly outperforms the random approach. The exceptions in the *Error* column reflect the already above mentioned preference of the execution time in our fitness function, which is further highlighted by the *Time* column. For instance, in the worst case (the *Crawler* test case), our genetic approach is more than 3 times faster but in total discovers three times less errors. On the other hand, in the best cases (the *Airlines* and *Rover*), we found three times more errors in three times shorter time. To give some idea about the needed time in total numbers, the average time needed to evaluate 2000 random individuals took on average 32 hours (whereas the genetic approach needed just 10.5 hours), and the average time needed to evaluate 2000 random individuals of our biggest test case *FTPServer* took 101 hours (whereas the genetic approach needed on average just 53 hours).

Overall, our results show that our approach outperforms the random approach. They also indicate that we should probably partially reconsider our fitness function that puts sometimes too much stress on the execution time, which can in some cases (demonstrated in the *Crawler* test case) be counter-productive.

Another positive fact is that our objective function helps to improve the testing process even for test cases that do not contain a data race. This can be attributed to that our fitness favours configurations within which the synchronisation occurs more often and therefore is tested more. The results obtained from our experiment with the *Crawler* test case evaluated using two different hardware configurations indicate that the genetic approach is able to reflect the environment and focus on the noise heuristics and their parameters which provide better results for the considered environment.

5.8 Summary

In this chapter, we have introduced and formulated the test and noise configuration search (TNCS) problem. We have conducted experiments with the simple Hill-climbing algorithm which was able to overcome the random approach in some cases. However, the experiments also showed that a local search technique does not need to be suitable for solving the TNCS problem. Based on the obtained experience, we have proposed way how to use genetic algorithms to solve the TNCS problem. We have performed experiments aimed at choosing suitable parameters of genetic algorithms to be used when solving the problem. We have instantiated the framework for the case of noise injection techniques implemented in the ConTest tool and its extensions and proposed a complex objective function suitable when aiming at data race detection (but successful even when looking for other kinds of bugs). We have performed experiments on a set of benchmark programs showing that our approach significantly outperforms the commonly used approach of randomly selecting noise configurations.

Chapter 6

Conclusions and Future Directions

In this thesis, we have concentrated on concurrency-related errors and noise injection techniques that help to examine different thread interleavings during testing and dynamic analysis of concurrent programs and to detect even rarely manifesting errors. Apart from the presented taxonomy of concurrency-related errors, our main contribution can be divided into three parts concerning concurrency coverage metrics, noise injection heuristics, and a use of metaheuristics in noise-based testing. Each of these parts has been presented in a separate chapter and has its own summary. Let us now just shortly summarise all the results of the thesis once again followed by a discussion of future research directions.

6.1 Summary

The first part of our contribution is a methodology of deriving new *coverage metrics* from dynamic (and possibly also static) analyses designed for discovering bugs in concurrent programs. Using this idea, we have derived several new concrete metrics. These metrics capture important features of the behaviour of concurrently executing threads. Therefore, they are suitable for debugging and testing of concurrent programs. We have performed an empirical evaluation of these metrics, which has shown that several of them are indeed better for use in saturation-based and search-based testing than various previously known metrics.

In the next part of the thesis, we have proposed a new original *noise injection heuristics* which uses concurrency coverage information to decide where to put noise. We have used selected metrics (namely, the Avio$*$ and HBPair* coverage metrics and the error manifestation ratio) to compare multiple noise seeding and noise placement heuristics including the two versions of our newly proposed heuristics. We have shown that our newly proposed heuristics wins over the existing ones in some cases. Moreover, we have also presented a systematic comparison of the previously known heuristics (which has missing in the literature) and shown that often a combination of several heuristics provide better results than any single heuristics.

As the last part of our main contribution, we have proposed a way of using search techniques to improve quality of noise-based testing and dynamic analysis through finding suitable combinations of parameters of tests and noise heuristics. We have formalised this problem as the *test and noise configuration search (TNCS) problem*. We have conducted experiments with the simple Hill-climbing algorithm. Based on the obtained experiences, we have proposed a way how to use genetic algorithms to solve the TNCS problem. We have performed experiments aimed at choosing suitable parameters of genetic algorithms to be used when solving the problem. We have proposed a complex objective function suitable for data race detection which is based on a dynamic analysis algorithm (namely, the GoldiLocks algorithm). Our experiments has shown that the objective function has been also successful at looking for other kinds of concurrency bugs. We have shown on a set of benchmark programs that our approach significantly outperforms the commonly used approach of randomly selecting noise configurations.

Besides the described three contributions, we have presented a uniform classification of common concurrency errors and provided a brief overview of various run-time verification and advanced testing techniques.

6.2 Future Research Directions

Future research in the area related to the thesis may be divided to the four categories discussed below.

Noise injection techniques. Current noise placement heuristics are rather simple and depend on simple code patterns and in most cases on a random generator. It was shown that even these primitive heuristics can improve the testing process. However, works on deterministic testing demonstrate that testing can efficiently use principles known in model checking and other formal approaches. Likewise, our results with coverage-based noise placement heuristics indicate that it is possible to efficiently use relatively complex heuristics which can beat random-based heuristics even in the question of performance. Hence, there is definitely a space for proposing new and better problem-specific noise injection techniques.

Search-based testing techniques. Our formulation of the TNCS problem allows one to use various metaheuristic optimisation algorithms in testing of concurrent programs. In this thesis, we applied only basic Hill-climbing and genetic algorithms to the problem. There are, however, advanced metaheuristic approaches which should provide even better results than the basic algorithms we used. Morover, even the use of genetic algorithms could be further improved by designing better objective functions which eliminate the negative effect we saw in the *Rover* test case in Chapter 5, which focus on different kinds of concurrency errors (e.g., deadlocks), or which are able to focus on corner cases of the tests. The TNCS problem is rather multi-objective because one would like to minimise the execution time and maximise the error detection probability. Therefore, we suggest to focus on approaches suitable for multi-objective problems.

Incorporation of static analyses. In the thesis, we used static analysis during instrumentation of the test cases to discover concurrency-related events considered by the IBM Concurrency Testing tool only. Most of those events actually represent accesses to heap objects which are not shared among threads. Putting noise before/after such events is therefore ineffective. Various static analyses could be used to decrease the number of events to be targeted by the noise and rapidly improve the performance of our approach. Static analysis can also be used to pinpoint variables, code patterns, and events which should be focused by the noise.

Data mining. We have created a database of test results including various concurrency coverage data. So far, we used the data mining approach only to discover outliers, i.e., values that are distant from the rest of obtained results, which might indicate a problem in our infrastructure. However, various data mining techniques can be used to further study the captured behaviour and the effects of various noise injection techniques. For instance, one can focus on the relation among the captured coverage tasks and error manifestation in hope of discovering an evidence which can be used for further debugging.

6.3 Publications Related to This Work

The taxonomy of concurrency errors presented in Chapter 2 appeared in [38]. The full version of the paper was published as the technical report [37]. The concurrency coverage metrics presented in Chapter 3 were published in [62], and the comparison of noise injection techniques included in Chapter 4 in [63].

The search-based testing of concurrent programs approach discussed in Chpater 5 was first proposed in [68]. The experiments with the Hill-climbing algorithm were published together with description of the SearchBestie infrastructure in [60]. The experiments with the genetic algorithms were accepted to appear in [55] and the full version of the paper was published as the technical report [54].

Results presented in this thesis build on the experience and technology that we in our previous works utilising the noise injection techniques [58, 67, 59, 61]. In [58, 59, 67], published within the MSc studies of the author, we studied how the noise techniques can be used to prevent concurrency errors from manifestation and showed that the noise techniques affects execution of the test on different architectures differently. In the work [61] published in the first year of the authors' PhD studies, we presented the IBM Concurrency Testing Tool and our plug-ins for this tool. We used the same infrastructure to implement concurrency coverage generators and other dynamic detection algorithms exploited in this thesis.

Bibliography

[1] Concurrency Stress Test. Online at: *http://msdn.microsoft.com/en-us/library /windows/hardware/hh454184(v=vs.85).aspx*, August 2011.

[2] R. Affeldt and N. Kobayashi. A Coq Library for Verification of Concurrent Programs. *Electronic Notes in Theoretical Computer Science*, volume 199, pages 17–32, February 2008.

[3] W. Afzal, R. Torkar, and R. Feldt. A Systematic Review of Search-based Testing for Non-functional System Properties. *Information Software Technology*, volume 51, pages 957–976, June 2009.

[4] R. Agarwal and S. D. Stoller. Run-time Detection of Potential Deadlocks for Programs with Locks, Semaphores, and Condition Variables. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'06, pages 51–60, New York, NY, USA, 2006. ACM Press.

[5] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, second edition, 2006.

[6] E. Alba and F. Chicano. Finding Safety Errors with ACO. In *Proc. of Genetic and Evolutionary Computation*—GECCO'07, pages 1066–1073, New York, NY, USA, 2007. ACM Press.

[7] E. Alba and F. Chicano. Searching for Liveness Property Violations in Concurrent Systems with ACO. In *Proc. of Genetic and Evolutionary Computation*—GECCO'08, pages 1727–1734, New York, NY, USA, 2008. ACM Press.

[8] E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms. In *Proc. of Genetic and Evolutionary Computation*—GECCO'08, pages 1735–1742, New York, NY, USA, 2008. ACM Press.

[9] G. R. Andrews. *Concurrent Programming: Principles and Practice*. Benjamin-Cummings Publishing Corporation, 1991.

[10] H. Arndt, M. Bundschus, and A. Naegele. Towards a Next-generation Matrix Library for Java. In *Proc. of Computer Software and Applications Conference*—COMPSAC'09, pages 460–467, Washington, DC, USA, 2009. IEEE Computer Society.

[11] C. Artho, K. Havelund, and A. Biere. High-level Data Races. In *Proc of Verification and Validation of Enterprise Information Systems*—VVEIS'03, pages 82-93, Angers, France, 2003. ICEIS Press.

[12] C. Baier and J.-P. Katoen. *Principles of Model Checking*. The MIT Press, 2008.

[13] M. Ben. *Principles of Concurrent and Distributed Programming*. Addison-Wesley, second edition, 2006.

[14] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise Makers Need To Know Where To Be Silent – Producing Schedules That Find Bugs. In *Proc. of International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*—ISOLA'06, pages 458–465, Washington, DC, USA, 2006. IEEE Computer Society.

[15] Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Producing Scheduling That Causes Concurrent Programs To Fail. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'06, pages 37–40, New York, NY, USA, 2006. ACM Press.

[16] Y. Ben-Asher, E. Farchi, and Y. Eytani. Heuristics for Finding Concurrent Bugs. In *Proc. of International Symposium on Parallel and Distributed Processing*—IPDPS'03, pages 288–304, Washington, DC, USA, 2003. IEEE Computer Society.

[17] S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'05, Haifa, Israel, volume 3875 of LNCS, pages 208–223, 2005. Springer-Verlag.

[18] Y. Bertot and P. Casteran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.

[19] J. Blieberger, B. Burgstaller, and R. Mittermayr. Static Detection of Livelocks in Ada Multitasking Programs. In *Proc. Reliable Software Technologies*—Ada-Europe'07, Geneva, Switzerland, volume 4498 of LNCS, pages 69–83, 2007. Springer-Verlag.

[20] E. Bodden and K. Havelund. Racer: Effective Race Detection Using AspectJ. In *Proc of International Symposium on Software Testing and Analysis–* ISSTA'08, pages 155–166, New York, NY, USA, 2008. ACM Press.

[21] J. S. Bradbury and K. Jalbert. Defining a Catalog of Programming Antipatterns for Concurrent Java. In *Proc. of Software Patterns and Quality*—SPAQu'09, pages 6–11, 2009. GRACE Center.

[22] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc of Principles and Practice of Parallel Programming*—PPoPP'05, pages 206–212, New York, NY, USA, 2005. ACM Press.

[23] F. Cabe. *Let's Go!*. Lulu.com, 2007.

[24] R. H. Carver and K.-C. Tai. *Modern Multithreading: Implementing, Testing, and Debugging Multithreaded Java and C++/Pthreads/Win32 Programs*. Wiley-Interscience, 2005.

[25] F. Chicano, M. Ferreira, and E. Alba. Comparing Metaheuristic Algorithms for Error Detection in Java Programs. In *Proc. of Search-based Software Engineering*—SSBSE'11, Szeged, Hungary, volume 6956 of LNCS, pages 82–96, 2011. Springer-Verlag.

[26] J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and Precise Datarace Detection for Multithreaded Object-oriented Programs. In *Proc of the Programming Language Design and Implementation*—PLDI'02, pages 258–269, New York, NY, USA, 2002. ACM Press.

[27] J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, *et al.* Reformulating Software Engineering as a Search Problem. *IEE Proceedings – Software*, volume 150, pages 161–175, June 2003.

[28] E. G. Coffman, M. Elphick, and A. Shoshani. System Deadlocks. *ACM Computer Surveys*, volume 3, pages 67–78, June 1971.

[29] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. of Principles of Programming Languages*–POPL'77, pages 238–252, Los Angeles, California, 1977. ACM Press.

[30] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, volume 15, pages 485–499, January 2003.

[31] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, volume 41, pages 111–125, January 2002.

[32] T. Elmas, S. Qadeer, and S. Tasiran. GoldiLocks: A Race and Transaction-aware Java Runtime. In *Proc. of Programming Language Design and Implementation*—PLDI'07, pages 245–255, New York, NY, USA, 2007. ACM Press.

[33] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The Notions of Consistency and Predicate Locks in a Database System. *Communications ACM*, volume 19, pages 624–633, November 1976.

[34] Y. Eytani. Concurrent Java Test Generation as a Search Problem. *Electronic Notes of Theoretical Computer Science*, volume 144, pages 57–72, May 2006.

[35] Y. Eytani and T. Latvala. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. In *Proc. of Haifa Verification Conference*—HVC'06, Haifa, Israel, volume 4383 of LNCS, pages 183–197, 2007. Springer-Verlag.

[36] E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How to Test Them. In *Proc. of International Symposium on Parallel and Distributed Processing*—IPDPS'03, pages 286–293, Washington, DC, USA, 2003. IEEE Computer Society.

[37] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar. A Uniform Classification of Common Concurrency Errors. Technical report FIT-TR-2010-03, Brno University of Technology, Czech Republic, 2010.

[38] J. Fiedor, B. Křena, Z. Letko, and T. Vojnar. A uniform classification of common concurrency errors. In *Proc. of Computer Aided Systems Theory*—EUROCAST'11, Las Palmas, Spain, volume 6927 of LNCS, pages 519–526, 2012. Springer-Verlag.

[39] C. Flanagan and S. N. Freund. Atomizer: A Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of Principles of Programming Languages*—POPL'08, pages 256–267, New York, NY, USA, 2004. ACM Press.

[40] C. Flanagan and S. N. Freund. FastTrack: Efficient and Precise Dynamic Race Detection. In *Proc. of Programming Language Design and Implementation*—PLDI'09, pages 121–133, New York, NY, USA, 2009. ACM Press.

[41] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: A Sound and Complete Dynamic Atomicity Checker for Multithreaded Programs. In *Proc. of Pro-*

gramming Language Design and Implementation—PLDI'08, pages 121–133, New York, NY, USA, 2008. ACM Press.

[42] D. Giannakopoulou, C. S. Pasareanu, M. Lowry, and R. Washington. Lifecycle Verification of the Nasa Ames K9 Rover Executive. In *Proc. of Verification and Validation of Model-Based Planning and Scheduling Systems*—ICAPS'05:, pages 11, 2005. AAAI Press.

[43] P. Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems: An Approach to the State-Explosion Problem.* Springer-Verlag New York, 1996.

[44] P. Godefroid. Software Model Checking: The Verisoft Approach. *Formal Methods in System Design*, volume 26, pages 77–101, March 2005.

[45] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, volume 6, pages 117–127, August 2004.

[46] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification.* Addison-Wesley Professional, third edition, 2005.

[47] C. Hammer, J. Dolby, M. Vaziri, and F. Tip. Dynamic Detection of Atomic-set-serializability Violations. In *Proc. of International Conference on Software Engineering*—ICSE'08, pages 231–240, New York, NY, USA, 2008. ACM Press.

[48] M. Harman and P. McMinn. A Theoretical and Empirical Study of Search-based Testing: Local, Global, and Hybrid Search. *IEEE Transactions on Software Engineering*, volume 99, pages 226–247, March 2009.

[49] K. Havelund. Using Runtime Analysis to Guide Model Checking of Java Programs. In *Proc. of Model Checking and Software Verification*—SPIN'00, London, UK, volume 1885 of LNCS, pages 245–264, 2000. Springer-Verlag.

[50] K. Havelund and A. Goldberg. Verify Your Runs. In *Proc. of Verified Software: Theories, Tools, Experiments*—VSTTE'05, Zürich, Switzerland, volume 4171 of LNCS, pages 374–383, 2005. Springer-Verlag.

[51] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming.* Morgan Kaufmann, 2008.

[52] A. Ho, S. Smith, and S. Hand. On Deadlock, Livelock, and Forward Progress. Technical report UCAM-CL-TR-633, University of Cambridge, 2005.

[53] G. Holzmann. *The Spin Model Checker: Primer and Reference Manual.* Addison-Wesley Professional, 2003.

[54] V. Hrubá, B. Křena, Z. Letko, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. Technical report FIT-TR-2012-01, Brno University of Technology, 2010.

[55] V. Hrubá, B. Křena, Z. Letko, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. Accepted for publication at *the 4th Symposium on Search Based Software Engineering*—SSBSE'12, Trento, Italy, 2012.

[56] J. Huang, P. Liu, and C. Zhang. Leap: Lightweight Deterministic Multi-processor Replay of Concurrent Java Programs. In *Proc. of Foundations of Software Engineering*—FSE'10, pages 207–216, New York, NY, USA, 2010. ACM Press.

[57] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proc. of Programming Language Design and Implementation*—PLDI'09, pages 110–120, New York, NY, USA, 2009. ACM Press.

[58] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-the-fly. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'07, pages 54–64, New York, NY, USA, 2007. ACM Press.

[59] B. Křena, Z. Letko, and T. Vojnar. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'08, pages 1–10, New York, NY, USA, 2008. ACM Press.

[60] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'10, pages 48–58, New York, NY, USA, 2010. ACM Press.

[61] B. Křena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref-Brill, S. Ur, and T. Vojnar. A Concurrency Testing Tool and Its Plug-ins for Dynamic Analysis and Runtime Healing. In *Proc. of Runtime Verification*—RV'09, Grenoble, France, volume 5779 of LNCS, pages 101–114, 2009. Springer-Verlag.

[62] B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of Runtime Verification*—RV'11, San Francisco, CA, USA, volume 7186 of LNCS, pages 177–192, 2012. Springer-Verlag.

[63] B. Křena, Z. Letko, and T. Vojnar. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of Mathematical and Engineering Methods in Computer Science*—MEMICS'11, Lednice, Czech Republic, volume 7119 of LNCS, pages 123–131, 2012. Springer-Verlag.

[64] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications ACM*, volume 21, pages 558–565, January 1978.

[65] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley. 2000.

[66] Y. Lei and R. H. Carver. Reachability Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, volume 32, pages 382–403, June 2006.

[67] Z. Letko. Dynamic Detection and Healing of Data Races in Java. Master's thesis, FIT, Brno University of Technology, 2008.

[68] Z. Letko. Sophisticated Testing of Concurrent Software. In *Proc. of Symposium on Search-based Software Engineering*—SSBSE'10, pages 36–40, Benevento, Italy, 2010. IEEE Computer Society.

[69] S. Leue, A. Ştefănescu, and W. Wei. A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems. In *Proc. of Concurrency Theory*—CONCUR'06, Bonn, Germany, volume 4137 of LNCS, pages 79–94, 2006. Springer-Verlag.

[70] R. J. Lipton. Reduction: A Method of Proving Properties of Parallel Programs. *Communications ACM*, volume 18, pages 717–721, January 1975.

[71] B. Long and P. Strooper. A Classification of Concurrency Failures in Java Components. In *Proc. of International Symposium on Parallel and Distributed Processing*—IPDPS'03, pages 287–295, Washington, DC, USA, 2003. IEEE Computer Society.

[72] S. Lu, W. Jiang, and Y. Zhou. A Study of Interleaving Coverage Criteria. In *Proc. of European Software Engineering Conference*—ESEC-FSE'07, pages 533–536, New York, NY, USA, 2007. ACM Press.

[73] S. Lu, S. Park, C. Hu, X. Ma, W. Jiang, Z. Li, R. A. Popa, and Y. Zhou. MUVI: Automatically Inferring Multi-Variable Access Correlations and Detecting Related Semantic and Concurrency Bugs. In *Proc. of Symposium on Operating Systems Principles*—SOSP'07, pages 103-116,New York, NY, USA, 2007. ACM Press.

[74] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning From Mistakes: A Comprehensive Study on Real World Concurrency Bug Characteristics. In *Proc. of Architectural Support for Programming Languages and Operating Systems*—ASPLOS'08, pages 329–339, New York, NY, USA, 2008. ACM Press.

[75] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations Via Access Interleaving Invariants. In *Proc. of Architectural Support for Programming Languages and Operating Systems*—ASPLOS' 06, pages 37–48, New York, NY, USA, 2006. ACM Press.

[76] S. Luke. *Essentials of Metaheuristics*. Lulu, 2009. Available at http://cs.gmu.edu/∼sean/book/metaheuristics/.

[77] J. Magee and J. Kramer. *Concurrency – State Models and Java Programs*. Wiley, second edition, 2006.

[78] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, 1992.

[79] J. Manson, W. Pugh, and S. V. Adve. The Java Memory Model. In *Proc. of Principles of Programming Languages*—POPL'05, pages 378–391, New York, NY, USA, 2005. ACM Press.

[80] F. Mattern. Virtual Time and Global States of Distributed Systems. In *Proc. of International Workshop on Parallel and Distributed Algorithms*—PDAA' 88, pages 215–226, 1988. North-Holland.

[81] P. McMinn. Search-based Software Test Data Generation: A Survey: Research Articles. *Software Testing Verification and Reliability*, volume 14, pages 105–156, December 2004.

[82] I. Mierswa, M. Wurst, R. Klinkenberg, M. Scholz, and T. Euler. YALE: Rapid Prototyping for Complex Data Mining Tasks. In *Proc. of Knowledge Discovery and Data Mining*—KDD'06, pages 935–940, New York, NY, USA, 2006. ACM Press.

[83] J. C. Mogul and K. K. Ramakrishnan. Eliminating Receive Livelock In an Interrupt-driven Kernel. *ACM Transactions on Computer Systems*, volume 15, pages 217–252, August 1997.

[84] S. Morasca and M. Pezze. Using High-level Petri Nets for Testing Concurrent and Real-time Systems. In *Proc. of Real-Time Systems: Theory and Applications*, pages 119–131, Amsterodam, Netherlands, 1990. North-Holland.

[85] M. Musuvathi and S. Qadeer. Iterative Context Bounding for Dystematic Testing of Multithreaded Programs. In *Proc. of Programming Language Design and Implementation*—PLDI'07, pages 446–455, New York, NY, USA, 2007. ACM Press.

[86] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. of*

Symposium on Operating Systems Design and Implementation—OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

[87] S. Narayanasamy, Z. Wang, J. Tigani, A. Edwards, and B. Calder. Automatically Classifying Benign and Harmful Data Races Using Replay Analysis. In *Proc. of Programming Language Design and Implementation*—PLDI'07, pages 22–31, New York, NY, USA, 2007. ACM Press.

[88] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, 1999.

[89] Y. Nir-Buchbinder, R. Tzoref, and S. Ur. Deadlocks: From Exhibiting to Healing. In *Proc. of Runtime Verification*—RV'08, Budapest, Hungary, volume 5289 of LNCS, pages 104–118, 2008. Springer-Verlag.

[90] Y. Nir-Buchbinder and S. Ur. ConTest Listeners: A Concurrency-oriented Infrastructure for Java Test and Heal Tools. In *Proc. of Software Quality Assurance*—SOQUA'07, pages 9–16, New York, NY, USA, 2007. ACM Press.

[91] Y. Nonaka, K. Ushijima, H. Serizawa, S. Murata, and J. Cheng. A Run-time Deadlock Detector for Concurrent Java Programs. In *Proc. of Asia-Pacific on Software Engineering Conference*—APSEC'01, pages 45–52, Washington, DC, USA, 2001. IEEE Computer Society.

[92] R. O'Callahan and J.-D. Choi. Hybrid Dynamic Data Race Detection. In *Proc. of Principles and Practice of Parallel Programming*—PPoPP'03, pages 167–178, New York, NY, USA, 2003. ACM Press.

[93] T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[94] A. Piziali. *Functional Verification Coverage Measurement and Analysis*. Springer-Verlag, 2007.

[95] E. Pozniansky and A. Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of Principles and Practice of Parallel Programming*—PPoPP'03, pages 179–190, New York, NY, USA, 2003. ACM Press.

[96] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of Operating Systems Principles*—SOSP'97, pages 27–37, New York, NY, USA, 1997. ACM Press.

[97] N. Shavit and D. Touitou. Software Transactional Memory. In *Proc. of Principles of Distributed Computing*—PODC'95, pages 204–213, New York, NY, USA, 1995. ACM Press.

[98] E. Sherman, M. B. Dwyer, and S. Elbaum. Saturation-based Testing of Concurrent Programs. In *Proc. of European Software Engineering Conference*—ESEC-FSE'09, pages 53–62, New York, NY, USA, 2009. ACM Press.

[99] J. Soriano, M. Jimenez, J. M. Cantera, and J. J. Hierro. Delivering Mobile Enterprise Services on Morfeo's MC Open Source Platform. In *Proc. of Mobile Data Management*—MDM'06, page 139–147, Washington, DC, USA, 2006. IEEE Computer Society.

[100] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice Hall, sixth edition, 2008.

[101] S. D. Stoller. Testing Concurrent Java Programs Using Randomized Scheduling. In *Proc. of Runtime Verification*—RV'09, *Electronic Notes in Theoretical Computer Science*, volume 70(4), pages 142–157, Elsevier. July 2002.

[102] V. Subramaniam and S. Venkat. *Programming Scala: Tackle Multi-core Complexity on the Java Virtual Machine*. Pragmatic Bookshelf, 2009.

[103] K.-C. Tai. Definitions and Detection of Deadlock, Livelock, and Starvation in Concurrent Programs. In *Proc. of International Conference on Parallel Processing*—ICPP'94, pages 69–72, Washington, DC, USA, 1994. IEEE Computer Society.

[104] E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.

[105] A. S. Tanenbaum. *Modern Operating Systems*. Prentice Hall Press, 2007.

[106] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, volume 18, pages 206–215, March 1992.

[107] 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 *Proc. of Parallel and Distributed Systems: Testing and Debugging*—PADTAD'09, pages 1–6, New York, NY, USA, 2009. ACM Press.

[108] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of International Symposium on Software Testing and Analysis*—ISSTA'07, pages 27–38, New York, NY, USA, 2007. ACM Press.

[109] M. Vaziri, F. Tip, and J. Dolby. Associating Synchronization Constraints with Data in an Object-oriented Language. In *Proc. of Principles of Programming*

Languages—POPL'06, pages 334–345, New York, NY, USA, 2006. ACM Press.

[110] C. von Praun and T. R. Gross. Object Race Detection. In *Proc. of Object Oriented Programming, Systems, Languages, and Applications*—OOPSLA'01, pages 70–82, New York, NY, USA, 2001. ACM Press.

[111] J. Šimša, R. Bryant, and G. Gibson. dbug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Proc. of Model Checking and Software Verification*—SPIN'00, London, UK, volume 1885 of LNCS, pages 188–193, 2000. Springer-Verlag.

[112] L. Wang and S. D. Stoller. Runtime Analysis of Atomicity for Multithreaded Programs. *IEEE Transactions on Software Engineering*, volume 32(2), pages 93–110, February 2006.

[113] E. J. Weyuker. The Evaluation of Program-based Software Test Data Adequacy Criteria. *Communications ACM*, volume 31, pages 668–675, June 1988.

[114] D. White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, volume 13, pages 65–67, March 2012.

[115] M. Xu, R. Bodík, and M. D. Hill. A Serializability Violation Detector for Shared-memory Server Programs. *SIGPLAN Notices*, volume 40(6), pages 1–14, June 2005.

[116] C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-du-path Coverage for Parallel Programs. In *Proc. of International Symposium on Software Testing and Analysis*—ISSTA'98, pages 153–162, New York, NY, USA, 1998. ACM Press.

[117] J. Yu and S. Narayanasamy. A Case for an Interleaving Constrained Shared-memory Multi-processor. *Computer Architecture News*, volume 37(3), pages 325–336, June 2009.

[118] Y. Yu, T. Rodeheffer, and W. Chen. RaceTrack: Efficient Detection of Data Race Conditions via Adaptive Tracking. *Operating Systems Review*, volume 39(5), pages 221–234, December 2005.

[119] W. Zhang, C. Sun, and S. Lu. ConMem: Detecting Severe Concurrency Bugs Through an Effect-oriented Approach. In *Proc. of Architectural Support for Programming Languages and Operating Systems*—ASPLOS'10, pages 179–192, New York, NY, USA, 2010. ACM Press.

Appendix A

Experiments with Parameters of Genetic Algorithms

This appendix presents selected results we obtained when experimenting with different values of parameters of the genetic algorithms presented in Chapter 5. Our motivation for these experiments was to discover how the different values of these parameters influence the performance of the genetic algorithm when solving the test and noise configuration setting (TNCS) problem formalised in Chapter 5. In particular, we studied the performance of the breeding process on the Crawler test case (introduced in Chapter 3) using a fitness function which combines three concurrency-related coverage metrics introduced in Chapter 3, namely, *Synchro*, *Avio** and *HB-Pair**. From each experiment, we collected various data concerning the generated populations including, in particular, the following three statistics: (i) The average fitness value in each generation denoted as *aver*, (ii) the best individual fitness in each generation denoted as *best*, and (iii) the cumulative value of fitness from all already evaluated individuals denoted as *gcumul*. Further details concerning our testing environment can be found in Section 5.5.

Results of three analyses are presented here. First, in Section A.1, we focus on settings of the population size (200, 80, 40, 20, and 10), the crossover operator(*one-point*, *two-point*, and *any-point* with the probability of mutating each element of the vector set to 0.1 or 0.25), and the probability of applying the mutation operator (0.01, 0.1, 0.25, and 0.5). Then, in Section A.2, we study the influence of elitism which puts into the next generation a number of individuals (0, 2, 4) without breeding, and a random creation of a few individuals (0, 2, 4) that are put into the following generation. In the third set of experiments presented in Section A.3, we focus on the influence of different selection operators (fitness proportional and tournament selection operators—with the size of the tournament being 2 or 4). A summary of all the results is provided in Section 5.5. The considered constants (e.g., sizes of population and various probabilities) were chosen with respect to values suggested in the literature, e.g., [104].

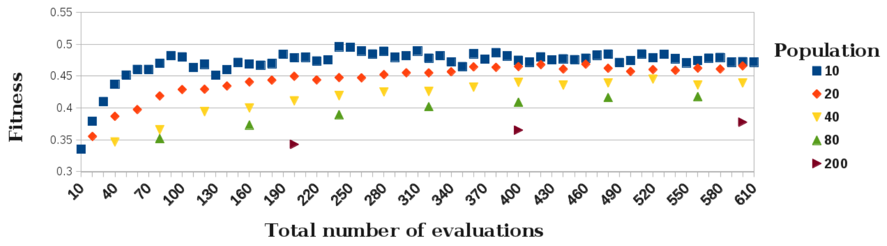


Figure A.1: Influence of the population size on the average fitness value within generations

A.1 Population Size, Mutation, and Crossover Parameters

Our findings are illustrated using a set of graphs showing influence of the considered parameters on the *aver*, *best*, and *gcumul* statistics computed from our fitness function introduced above. Graphs in Figures A.1 and A.2 show the influence of the considered parameters on the value of the *aver* statistics. Graphs in Figures A.3 and A.5 demonstrate the influence of the parameters on the value of the *best* statistics. Finally, Figure A.6 presents a graph showing the influence of different crossover and mutation settings on the *gcumul* statistics in the two selected sizes of population.

In Figure A.1, we can see a graph showing average values of the *aver* statistics for different size of populations when a certain number of individuals was evaluated. The average value is computed from all experiments with a particular population size using different crossover and mutation parameters. We can see that the average fitness value computed from the small populations increases faster, saturates earlier, and almost always is higher than the average fitness value computed from bigger populations if the total number of evaluated individuals given by the x-axis is considered. This is natural because small populations have usually a lower diversity and one can perform the breeding process more often than with the big populations. But, if the values of the *aver* statistics are compared by generations (each symbol in the graph represent the value computed after each generation—therefore there are only 3 values computed for the population of size 200), one can see that values obtained by the first, second, etc. generations are similar. A difference can be found in the saturation point. For instance, the population with 10 individuals stops rapidly growing after 5 generations while the population with 20 individuals reaches similar values after 7 generations. In general, the saturation point is harder to detect for the bigger generations.

The graph in Figure A.2 shows average values of the *aver* statistics for the population of size 40 when different mutation and crossover parameters are used. The graph is divided into four parts each showing results for the particular setting of the

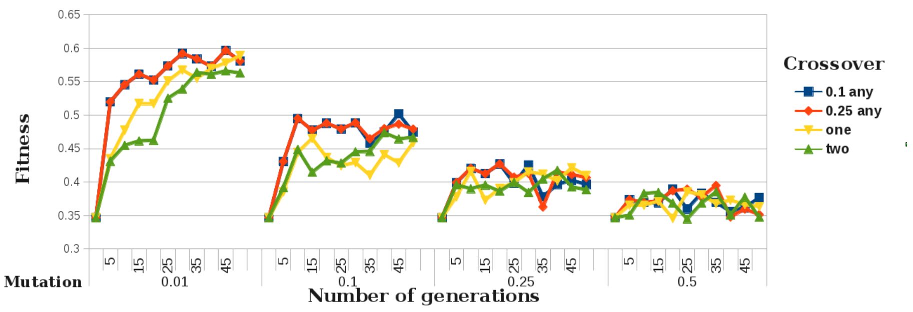


Figure A.2: Influence of the crossover and mutation parameters on the average fitness value within generations (population size 40)

mutation parameter. One can clearly see that lower mutation (i.e., the left parts of the graph) leads to higher values of the *aver* statistics. Again, this is natural hence higher values of the mutation probability introduce more diversity to the bread generation. The overall best results are obtained when the mutation probability is set to 0.01 and the *any-point* crossover operator is used. This crossover operator produces individuals which differ from their parents only a little if a low probability parameter is used. Overall, the smaller influence of the mutation and crossover parameters is, the higher value of the *aver* statistics is reached because the operators disturb the effect of the considered selection operators only a little.

The graph in Figure A.3 presents average values of the *best* statistics for different sizes of population with respect to the number of performed evaluations denoted as *global count*. Only the first 600 evaluations are zoomed for clearer overview. The average values for each population size are computed from all considered settings of the crossover and mutation parameters. At first, the values seem to be very similar for all the considered sizes of population. This is mainly due to considered average values we present here. Nevertheless, one can spot that the values achieved by the first generations of bigger populations are very similar (and in the case of population of size 20 and 200 even better) to values obtained by the configurations with lower sizes of population if the values of the same total number of evaluations is used. This means that even the random search algorithm provide comparative results for the given comparison.

The graph in Figure A.4 presents the same results as the previous graph presented in Figure A.3 but values from 1,000 to 2,000 evaluations are zoomed this time. The graph shows that within several generations, the biggest and smallest populations are overtaken by the middle sized populations. In the case of population of size 10, this is probably caused by reaching a local optimum in some experiments. In the case of population of size 200, the worse values are probably caused by the low number of generations which are not enough to reach the optimum as often as in the cases of populations with size 20, 40, and 80.

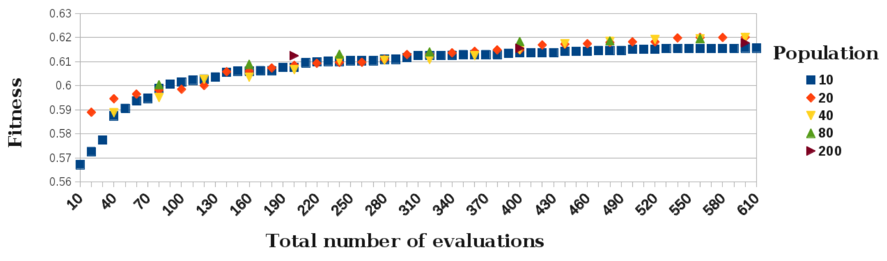


Figure A.3: Influence of the population size on the fitness value of the best individual showing only the first 600 evaluations

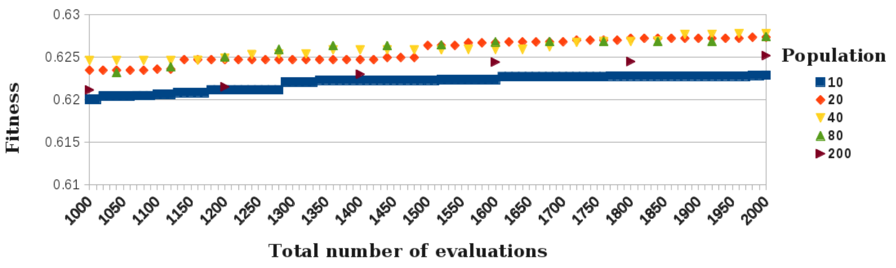


Figure A.4: Influence of the population size on the fitness value of the best individual showing only fragment between 1,000 and 2,000 evaluations

In Figure A.5, one can see a graph that shows the influence of the crossover (represented by colours) and mutation (represented by the four different parts of the graph) settings on the *best* statistics. The presented values are computed from results collected when the size of population was set to 20, but a very similar tendency can be seen for the other sizes of population as well. The results show that there is no clear winner among the configurations, and the differences among the settings are quite small, ranging from 0.62 to 0.63. The best individual was found with the highest probability when the mutation was set to 0.25 and the crossover was set to *any-point*. The setting which uses mutation with probability 0.25 seems to provide relatively good average values of the best individual within the first 15 generations regardless which crossover operator is used.

Finally, Figure A.6 presents a graph that shows the influence of the different settings of mutation and crossover parameters on the *gcumul* statistics. The graph shows the *gcumul* values after the first 240 evaluations of individuals (1200 executions of

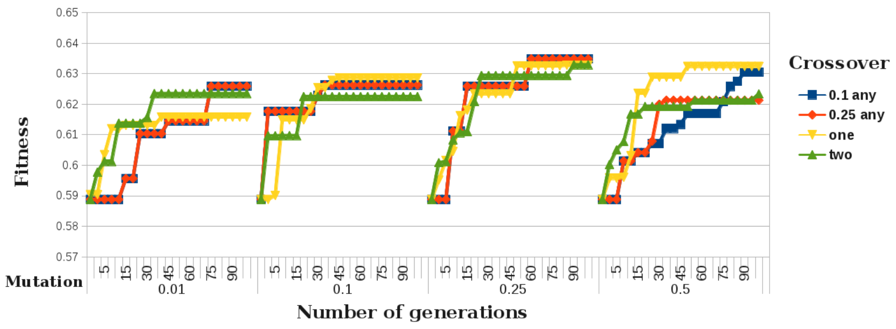


Figure A.5: Influence of the crossover and mutation parameters on the fitness value of the best individual (population size 20)

the test) for two sizes of populations (namely, 20 and 40). Therefore, we consider the values after 12 generations in the case of population size 20 and 6 generations in the case of population size 40. The differences among the considered settings are below 10 % of our fitness function. The graph indicates that the highest cumulative coverage during the beginning of the breeding process is on average achieved when the crossover operator is set to *any-point* and the mutation probability is set to 0.1. It can be also seen that in the case of the *any-point* crossover operator which changes the individuals only a little, the value of the mutation probability influences results more than in the case of the *two-point* crossover operator.

If we look at the progress of the *gcumul* statistics with different parameter settings, one can see that there is no silver bullet among the considered configurations. In nearly each generation, the highest value of the *gcumul* statistics is obtained using different sizes of the population and different settings of the other considered parameters. The situation changes after the saturation point which occurs after evaluating different numbers of generations for different parameter settings (often between 1000 and 1500 evaluations of individuals). After saturation, the results are very similar ranging from 0.88 to 0.92. In general, higher values after saturation were obtained using medium-sized populations (20, 40, and 80) with mutation probability 0.25 and the *one-point* or *two-point* crossover operators.

A.2 Elitism and Random Individuals

Next, we focus on the influence of the commonly used modification of genetic algorithms. In particular, on the influence of elitism (i.e., number of elite individuals that are added to the next generation without breeding) and random generation of individuals (i.e., number of randomly generated individuals that are added to the next

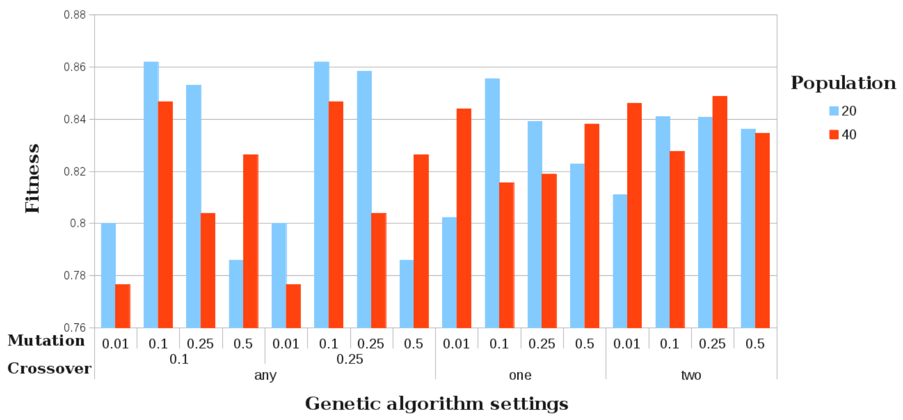


Figure A.6: Influence of the crossover and mutation parameters on the cumulative coverage (after 240 evaluations, population size 20 and 40)

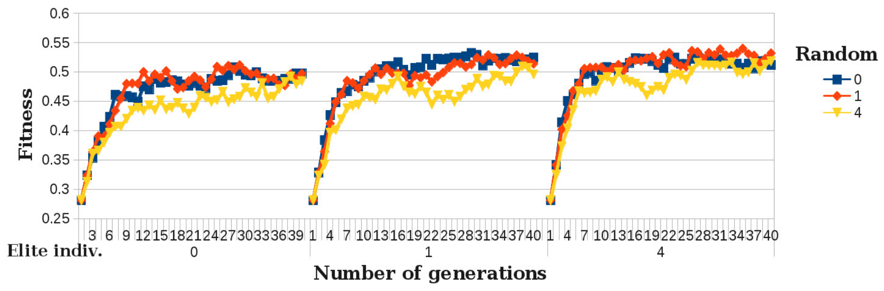


Figure A.7: Influence of elitism and random individuals on the average fitness value within generations

generation without breeding). The results presented here were achieved using the size of the population set to 40, the crossover operator set to *any-point* with probability 0.1, and the mutation probability set to 0.01. We considered three different numbers of elite and random individuals (namely, 0, 1, and 4) representing 0, 2.5 %, and 10 % of the population. Our results are summarised in three graphs depicted in Figures A.7, A.8, and A.9. The graphs show results for the first 40 generations only which capture the most interesting part of the breeding process.

The graph in Figure A.7 shows the influence of elitism represented by the 3 different parts of the graph and random generation of individuals represented by colours on the average fitness value within generations (the *aver* statistics). The values in the

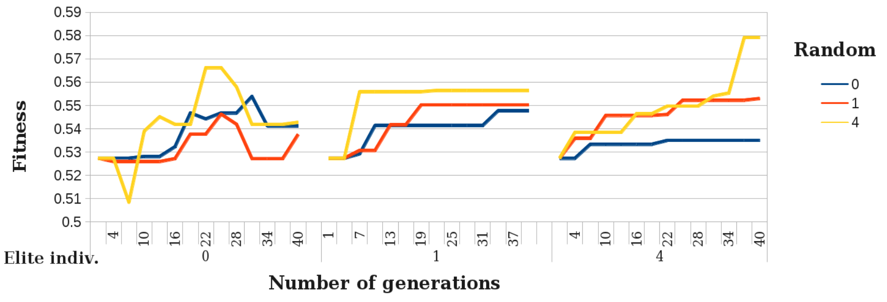


Figure A.8: Influence of elitism and random individuals on the best individuals

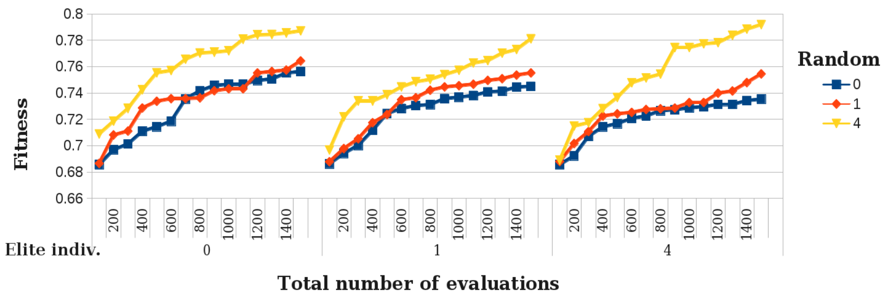


Figure A.9: Influence of elitism and random individuals on the cumulative coverage

graph were computed as average values obtained from 2 executions of each configuration. The influence of both parameters is evident. The presence of elite individuals has positive effect on the results with minimal difference between 1 and 4 elite individuals. And, the more randomness is present in the generations the lower average values are obtained.

The graph in Figure A.8 shows similar results as above for the value of the best individual (the *best* statistics). The leftmost part of the graph shows the situation when no elite individual is passed to the next generation without breeding. This has a negative effect on the *best* statistics which does not increase nicely. Instead, the values are jumping because in each generation, the best individual can achieve lower fitness than the best individual of the previous generation. However, in general, one can clearly see that the best individuals in the later generations achieve on average higher fitness than the best individuals in the first generations. All three parts of the graph demonstrate that the highest values of the fitness can be achieved when a high number of random individuals is present in each generation. One can also see small

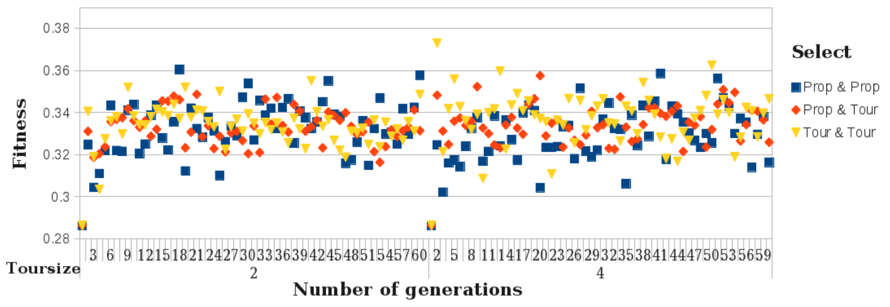


Figure A.10: Influence of the selection methods on the average fitness value

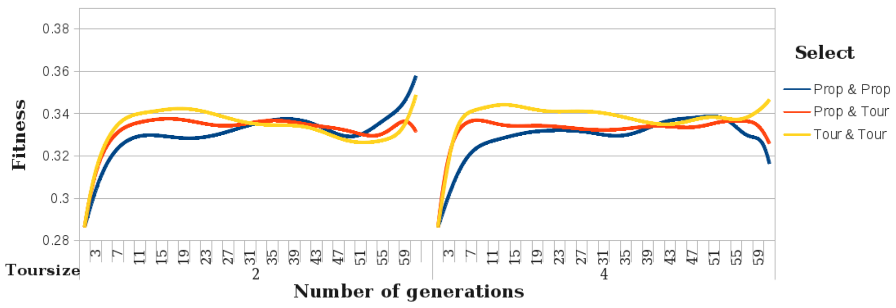


Figure A.11: Influence of the selection methods on the average fitness value – interpolated

but positive effect of the elitism. The overall best result is achieved by the setting with highest number of random and elite individuals.

In Figure A.9, a graph presenting the influence of the elite and random individuals on the cumulative coverage (the *gcumul* statistics) within the first 300 evaluations (40 generations) is shown. Again, the number of random individuals is represented by colours and the number of elite individuals divides the graph into three parts. One can clearly see that the more random individuals are present in each generation the higher cumulative coverage is reached. Two different tendencies concerning elite individuals can be seen here. If no or small random individuals are present, the elite individuals have a slightly negative impact on the cumulative coverage. Nevertheless, the overall best value was reached when both parameters were set to 10 % of the generation.

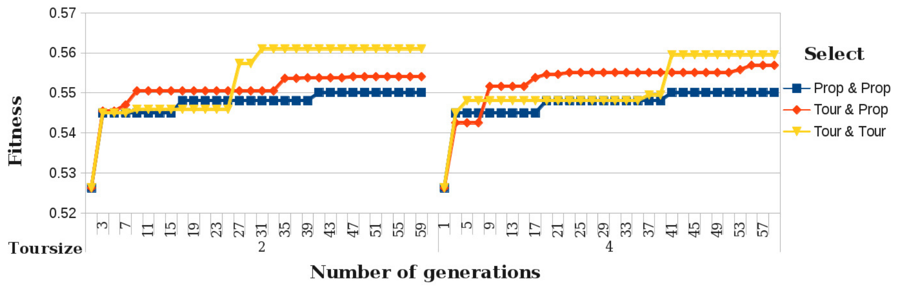


Figure A.12: Influence of the selection methods on the best individuals

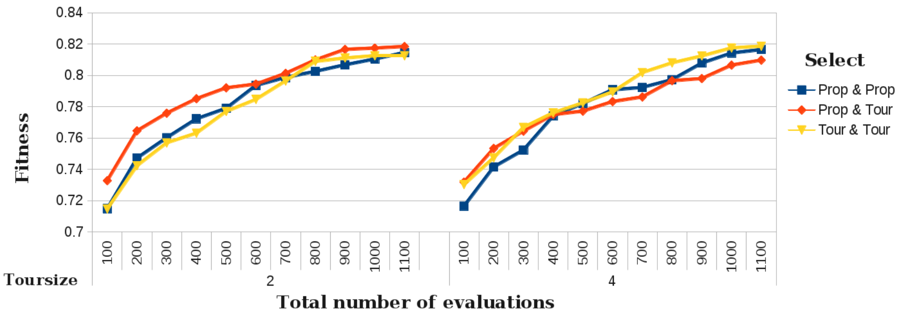


Figure A.13: Influence of the selection methods on the cumulative coverage

A.3 Selection Operators

Finally, we have focused on a comparison of different selection operators. In our breeding infrastructure, we use two selection operators which select two individuals that are then crossed over. We considered only two kinds of selection operators in our experiments. The *fitness proportional* operator (denoted as *prop* in the graphs) represents a roulette wheel selection operator [104]. The *tournament* operator (denoted as *tour* in the graphs) picks the individual with the highest value of the fitness out of a group of randomly selected individuals. We consider two sizes of the group (the tournament size)—2 and 4 individuals. Our findings are illustrated by the four graphs depicted in Figures A.10, A.11, A.12, and A.13. These results were obtained using a similar configuration as above. We set the size of population to 40, the crossover operator to *any-point* with probability 0.1, and the mutation probability to 0.01. We consider only one elite and no random individual in each generation.

The results for the *aver* statistics are not presented because the values are jumping among generations a lot as can be seen in Figure A.10. But, if we interpolate them as shown in Figure A.11, we clearly see that within the first 7 generations the value of the *aver* statistic rise (over approximately 10 %) and then stagnate. Overall, good results were obtained using combination two tournament operators and combination of the tournament and fitness proportional selection operators. However, the differences among various combinations were very low usually below 5 %.

The graph depicted in Figure A.12 shows the influence of the considered selection operators on the best individual. The graph is divided into two parts. The left part of the graph shows results for tournament size 2 and the right part of the graph shows results for the tournament size 4. Both cases are compared with the configuration based on two fitness proportional selection operators. The impact of different selection operators is minimal (around 1 %) as can be seen from the graph. This is probably caused by the similarity of the considered operators where all of them choose individuals having a promising fitness value with higher probability. But, it seems that the best results were obtained by the combination of the tournament (with tournament size 4) and fitness proportional selection operators. This combination provides the best results after a few first generations and at the end achieves very similar results to the best result achieved by the configuration where two tournament operators were used.

Finally, the graph presented in Figure A.13 illustrates the influence of the selection operators on the cumulative coverage. The graph is again divided into two parts as in the previous case. Results for the first 220 evaluations (1100 executions of the test) are presented. Again, a minimal influence of the selection operators can be seen. The best results were again achieved by the combination of fitness proportional and tournament selection (tournament size 2) operators.

Appendix B

An Infrastructure for Search-based Testing of Concurrent Programs

This appendix presents an infrastructure for search-based testing of concurrent programs which we developed and used in this thesis for an experimental evaluation of our ideas. The infrastructure is called *SearchBestie*. The name is an abbreviation for Search-Based Testing Environment. The infrastructure allows one to implement and experiment with various metaheuristic algorithms, and so it can be viewed as yet another tool for search-based optimisation. Indeed, the infrastructure can be used as a platform for various applications of metaheuristic optimisation algorithms to different problems including (but not limited to) program testing.

What makes the infrastructure different from its competitors is a support for testing of concurrent programs. This includes evaluation of candidate solutions using a test execution where the program under test is first instrumented, then executed together with a run-time analyse and a noise maker, and finally the results of the execution are gathered so that the fitness function can be evaluated. The second major difference is that a special attention is paid to specifics of concurrent programs testing introduced in Chapter 2 such as scheduling non-determinism. We reflect them mainly by various iterators which enable multiple execution of the same test together with checking a termination condition after each iteration and by providing combination operators which allow to aggregate results from multiple executions.

In fact, the goal was not to create a new general purpose tool for search-based testing but to create a glue which allows for cooperation among specialised tools for concurrency testing such as IBM Concurrency testing tool (ConTest) [61] described in Chapter 2 and tools or libraries which focus on metaheuristic optimisation algorithms such as the ECJ toolkit [114]. Therefore, a special attention is paid to an extensible design of our infrastructure. Moreover, SearchBestie can either be used as a tool for search-based testing or as a fitness evaluation procedure within another tool.

The secondary aim of the infrastructure is to create a database of obtained results which can be further studied. This includes collection and organisation of data produced by many executions of tests. The data can be of various forms including string identifiers, numbers, binary data, and coverage information describing both the configuration of the tests and the obtained results. SearchBestie allows to upload these results into a relational database. Then, further tools and techniques including *data-mining* techniques can be used to analyse the data.

The rest of the appendix is organised as follows. First, the extensible architecture of SearchBestie is presented in Section B.1. Then, the cooperation with other tools and libraries is described in Section B.2.

B.1 The Architecture of SearchBestie

The SearchBestie infrastructure is divided into four cooperating modules called the *Manager*, *State space storage*, *Search*, and *Executor*. The roles and interactions of all these modules are briefly introduced in the following paragraph. Each module is then described in more detail together with some design and implementation notes in the following four subsections.

A general overview of the structure and functioning of the SearchBestie architecture is provided in Figure B.1. The manager reads a configuration file and initialises other modules. Then, the manager enters a loop common for all search techniques. The manager asks the search engine to identify a state in the searched state space, which may be viewed as a test and its parameters, to be explored in the next step. The chosen state is then passed to the execution module that executes the appropriate test. Results of the test are collected, and an object encapsulating the results is passed back to the search engine as a feedback and stored in the state space storage. Subsequently, a test whether pre-defined termination conditions have been fulfilled is performed. If not, the next iteration starts, and the manager asks the search engine to provide a next state of the search space to be explored. When the searching is over, the manager can analyse the obtained results or export them.

The architecture is meant to be very generic and therefore all modules consist of two parts: an interface that communicates with the rest of the infrastructure and plug-ins that actually provide the functionality. Plug-ins can implement the functionality on their own or can implement an interface to an external library or tools. Since plug-ins for the same module share a common interface, they can be easily interchanged. This allows users to easily experiment with several different testing approaches. The generality of our architecture is also supported by the idea of building blocks that allow for combining several plug-ins into more complicated entities.

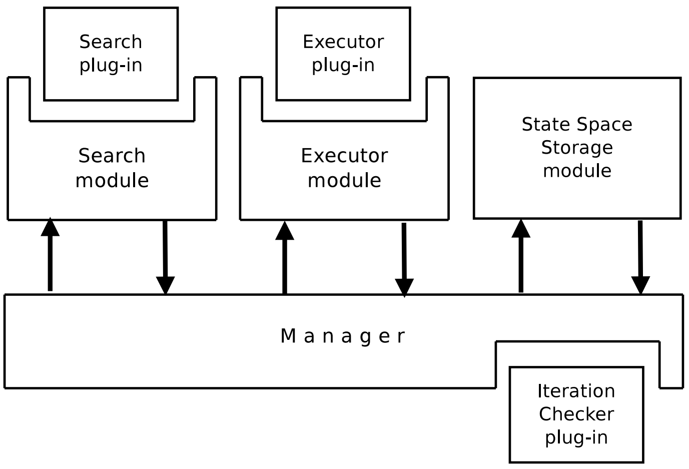


Figure B.1: High-level architecture of SearchBestie

The State Space Storage Module

The state space storage module encapsulates a model of the search space and stores and organises results obtained from the already explored states. The interface provided by this module allows other modules to get information concerning the search space and the results obtained so far.

The *search space* can have any number of dimensions, which may be of several different kinds. In particular, SearchBestie currently supports four types of dimensions: (i) boolean dimensions, (ii) integer dimensions with given minimal and maximal values, (iii) enumeration dimensions consisting of some number of values that can be strings, integers, and booleans, and, finally, (iv) *test dimensions*, which are special dimensions containing an enumeration of tests that are used to explore particular states.[1] Each dimension contains information whether values in the dimension are considered to be ordinal or not. Each state space must contain exactly one test dimension. So, if the test dimension contains only one test, the whole search space is explored using this test. However, for complex programs, we expect dozens of tests. We will describe tests in more detail in the following subsection.

A *state* in a search space is then a vector of values from all dimensions (one value per dimension), having the semantics of a test and its parameters. Of course, there can be states that are not allowed or are not interesting from the testing point of view (e.g., some parameter does not influence the particular test or some combination

[1]We use the name “test dimensions” as testing is the main application of SearchBestie for us. However, this does not restrict the use SearchBestie to testing only: The special “test” dimension can contain, in general, any functions to be run over particular states, leading to some evaluation of them, without any restriction of what is the real meaning of these functions.

of parameters needs not be valid for a particular test). Such combinations can be specified by a user-defined predicate. The search engine is notified when it tries to explore an illegal state.

When a state is explored, the executor engine produces a result that is stored in the state space storage module. The storage of results has the form of a map that maps already explored states to the corresponding results. The module can be configured to store only one result for each state or to store a set of results. When a particular state is explored multiple times, the user can define whether all results are stored or only one value obtained by applying a *combination operator* on the old and new result value. The storage module currently supports five combination operators, returning the better or worse (w.r.t. the result of a user defined fitness function), and accumulated, differentiated, or intersected results.

The *result* of a state exploration is a vector of values of various kinds and meanings. Each result contains at least two items in the vector that are introduced by SearchBestie. The first describes whether the test *passed*, *failed*, or was *interrupted*, and the second contains information on how long the exploration of the state took. Other values in the vector are considered to be problem-specific and left to the user to be defined. For instance, in our experiments, we often use results containing coverage information and the number of errors detected during the test execution.

When dealing with search-based algorithms, a computation of the *fitness function* is important. Since there could be several different fitness functions to be used over a vector of values in a result, we decouple the evaluation of fitness functions from the computation of results. Therefore, the user can experiment with different fitness functions without a need to change anything within the results.

The state space storage module can automatically provide various statistical data such as the number of so far explored states, the number of obtained results or the number of results with some specific value, coordinates of notable states (e.g., providing the highest value of the fitness function), cumulative results accumulated over the whole search space or over a selected subset of states, etc. The data can be exported as *views*, i.e., two dimensional tables which can be further analysed outside SearchBestie. The module can also export obtained results either to files on disk or to the SQL database. Exported data can be again imported or further analysed by external tools as described in Section B.2.

The Executor Module

The Executor module controls exploration of a chosen state from the state space either by executing an external program (e.g., a test) or by evaluating a function encoded into a Java class. Again, the module can be divided into an interface communicating with the rest of the infrastructure and a plug-in that actually performs the computation. As was mentioned in the previous subsection, the engine is determined by the value of the test dimension of the particular state being explored.

To allow users to easily create more complicated engines, a few adjustable building blocks are available. The building blocks are based on three kinds of engines: (i) A *task* performs some given work. Currently, tasks that execute processes or Java processes are available. (ii) A *test engine* is very similar to a task engine with the exception that when a test is performed, it generates a result. Therefore, the test engine actually implements exploration of a given state. (iii) An *iteration checker engine* takes as its parameter a generated result and decides whether a further exploration is needed or not.

Based on these basic engines, several more complex engines are available too: (i) A *timed test* encapsulates a given test engine and if an execution of the encapsulated test exceeds a given time bound, the execution is interrupted. (ii) A *repeated test* adds an iteration checker engine to a test. The test is then executed in a loop until a termination condition is fulfilled. One can also choose whether results generated by the encapsulated test are stored in the state space storage module or whether the resulting (best, worst, accumulated, etc.) result is stored there only. (iii) A *composed test* allows for adding additional tasks into the encapsulated test. The first task is optionally executed before performing the test and the second one is optionally executed after the test.

Since executing hundreds of tests is time demanding, we also implemented a simple plug-in that imports data from a previously explored state space. Afterwards, when the plug-in is asked to perform a test, a result obtained within the imported exploration or in the database is immediately returned. This way, the experiments can be performed quite fast, and the user can analyse results instantaneously as far as the configuration of the state space is the same and the plug-in is not asked to return more results than are available. In that case, the test is executed and results provided. This way, the infrastructure can be used to evaluate not yet evaluated states only and to share results among multiple instances of SearchBestie.

The Search Module

The search module consists of an interface which is used to cooperate with the rest of the infrastructure and a search algorithm implemented as a plug-in which actually does the search. The search module is called to provide the next state that should be explored. The search engine can access the state space storage module via an appropriate interface in order to get information concerning the search space (dimensions, results, statistics, etc.).

To prove the concept, we implemented several search algorithms: a sequential search that explores the whole search space in a sequential manner, a random search that explores randomly chosen (not yet explored) states, and the Hill-climbing search algorithm. We do not invest energy into creation of further and more complicated search algorithms because our goal was to allow SearchBestie to use external tools as described in Section B.2.

The Manager Module

The manager module is a service module that takes care of the initialisation and finalisation of SearchBestie, provides supporting tools for other modules (e.g., the logging subsystem), and during the exploration loop controls the flow of data among modules. The manager module also provides a simple interface for a communication with the user.

SearchBestie takes as input a few basic parameters (specifying, e.g., whether to import previous results or not, etc.) and an XML configuration file. The file contains a configuration for all modules and is divided into 3 parts (a configuration of the search module, the executor module, and the state space storage module). The main part of the configuration file is devoted to a definition of the dimensions of the search space to explore.

The algorithm of the search space exploration in SearchBestie was already mentioned in the beginning of this appendix. Let us, however, describe it in more detail now:

1. Optionally, an initialisation task is performed before the actual exploration begins. This task can be used to prepare tests, e.g., compile and/or instrument the code.

2. The search module is asked to identify a state to be explored. Each state has the test dimension that identifies the test engine to be used for an exploration of the state.

3. The chosen test engine is initialised and executed, taking as a parameter the state. The test generates a result.

4. The result generated by the test engine is passed to the search module as a feedback for the search engine and to the state space storage module to be stored.

5. Finally, the result is passed to the iteration checker engine that decides whether the termination condition is satisfied. If not, the execution continues by Step 2. If the termination condition is satisfied, the execution continues by the next step.

6. Optionally, a finalisation task is performed after the exploration. This task can be used to finalise tests, e.g., do some post-processing of the results, remove temporary files, etc.

There are multiple ways how the exploration loop may end. The exploration may be finished if a global timeout expires, the user makes the manager stop the exploration, the search engine does not provide any state to be explored, or the iteration checker executed at the end of each iteration detects that some pre-defined termination condition has been fulfilled. When the exploration loop is finished, the explored state space is exported to a file for further analysis. During the exploration, another output file is being produced. This file contains a report concerning the sequence of explored states and the corresponding achieved results (which may itself be interesting when exploring, e.g., how the accumulated coverage was increasing during the exploration).

When experimenting with different heuristics, one sometimes needs to perform multiple search space explorations and to compute average values of the obtained results in order to decrease influence of the non-determinism present in the used approach. Therefore, our manager module allows to define the number of times the process of exploration is performed and to collect important statistical data from such executions. Of course, it is also possible to subsequently analyse each exploration separately using logs and exported data.

B.2 A Tool Chain for Search-based Testing and Analysis

The extensible design of SearchBestie allows us to use the tool as a standalone program which performs a metaheuristic optimisation or as a glue interconnecting specialised tools and libraries which implement metaheuristic algorithms, dynamic analyses, a noise maker, etc. Moreover, the export functionality implemented in the state space storage module allows us to further transform, analyse, and visualise obtained results. This section describes in more details the way how SearchBestie communicates with tools which we used to obtain results presented in this thesis.

Figure B.2 presents the particular tool chain we used. The diagram on the figure shows interconnection of SearchBestie with other tools. Three major extensions of SearchBestie are captured in the figure:

i. The IBM Concurrency Testing Tool (ConTest) introduced in Chapter 2 is used by SearchBestie to perform test executions, inject noise, and collect results which are then used by SearchBestie to compute the corresponding fitness value. The communication with ConTest is realised within the executor module plug-in.

ii. The ECJ toolkit [114] provides evolutionary algorithms producing individuals to be evaluated by SearchBestie. In case of cooperation with ECJ, ECJ is executed first and SearchBestie is used to evaluate candidate solutions generated by ECJ.

iii. The state space storage module allows us to store the collected results into disk files, upload them into an SQL database, or produce various two dimensional views of the data which can be further processed by the Universal Java Matrix Package (UJMP) [10], any spreadsheet program, or further analysed by a data mining tool such as the RapidMiner (formerly called YALE) [82].

All three mentioned extensions are described in more details below.

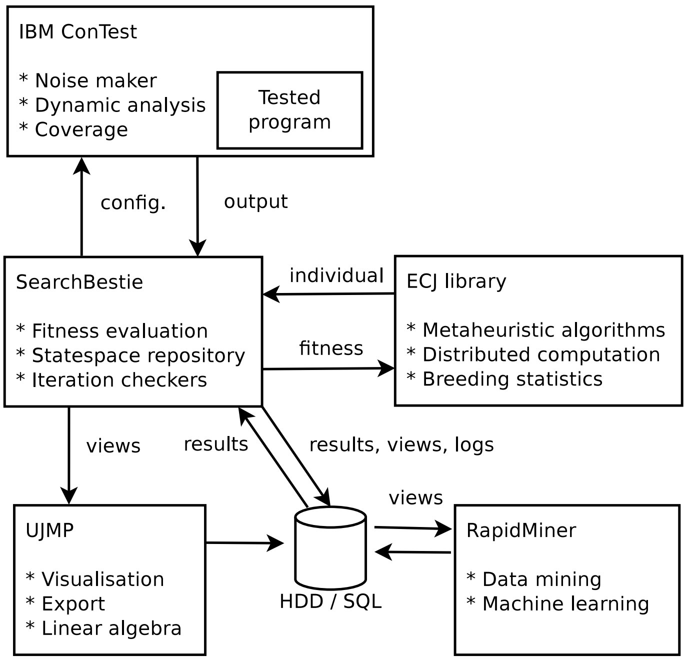


Figure B.2: A tool chain for concurrent program testing and analysis

Cooperation with the IBM ConTest

Typical use scenario of IBM ConTest is as follows: (i) First, Java byte-code of the test is instrumented before its execution. ConTest also provides dynamic instrumentation which instruments each class during its loading into memory. Moreover, since we often execute the same test many times, we prefer static instrumentation which is performed only once. (ii) When the instrumented code is executed, ConTest is initialised before executing the code of the test. During initialisation, ConTest reads its configuration files which contain parameter settings of ConTest, a list of enabled ConTest plug-ins, and parameters used by the plug-ins. ConTest also generates a *unique identifier* for the current execution. Then, the instrumented byte-code is executed. ConTest and its plug-ins produce outputs (e.g., the coverage) into the ConTest output directory. (iii) Next, the generated data from the execution are available in subdirectories of the output directory after execution. Each generated file contains the ConTest unique execution identifier in its name. (iv) Finally, unnecessary data produced by ConTest or already processed can be deleted.

The cooperation of SearchBestie with ConTest is implemented within the executor module plug-in and consists of two tasks implementing steps (i) and (iv) introduced above and one test engine which encapsulate steps (ii) and (iii). The only activity required by the user of SearchBestie is to enable this kind of test and properly set the parameters controlling its behaviour, the code to be executed, and parameters of ConTest and its plug-ins. The tasks can be activated by the test if enabled by corresponding parameters. One task is responsible for static instrumentation performed once before the search process within the SearchBestie begins. The their task is responsible for cleaning of the ConTest output directory after each evaluation of an individual.

The main functionality is implemented in the test engine which is responsible for generation of configuration files of ConTest and its plug-ins, execution of the test within a separate process, and importing generated data from the ConTest output directory into the vector of results used by SearchBestie to compute the fitness value. The configuration of the test engine can contain variables whose values are determined by SearchBestie according to the state of the state space currently being evaluated. The engine also allows to detect occurrence of exceptions by observing outputs of the executed test. Processing the executed test outputs also allows us to detect situations when the running test produces no output for a predefined time. This helps to detect deadlocks and some other progress problems. In such case, the execution can be terminated by the test engine. Currently, SearchBestie supports all functionality implemented in the latest version of IBM ConTest.

Cooperation with the ECJ Toolkit

ECJ is a Java-based evolutionary computation research system being developed for more than ten years. It supports a wide range of metaheuristic algorithms and approaches including genetic programming, genetic algorithms, evolutionary strategies, particle swarm optimisation, and differential evolution [114]. The toolkit have been chosen among many other existing tools and libraries mainly due to very well written documentation, well-arranged system of configuration files, and internal design which allows us to easily interconnect SearchBestie with ECJ.

The maturity of ECJ convinced us to redesign SearchBestie such that it allows external tools like ECJ to use SearchBestie as a procedure for evaluation of candidate solutions. The cooperation works as follows. ECJ is executed by the user and within the initialisation phase of ECJ, SearchBestie is also initialised. ECJ then generates individuals for evaluation and performs search. Each time ECJ requires an individual to be evaluated, SearchBestie is called. The evaluation consists of three steps: (i) First, the individual is transformed into the corresponding state in the state space used by SearchBestie. (ii) Then, the manager module evaluates the state as if it comes from the search module. (iii) Finally, the result is stored in the state space storage module and the computed fitness is passed back to ECJ. The search process can be stopped either by ECJ, e.g., when predefined number of generations is evaluated or by SearchBestie.

This design of SearchBestie allows us to use all features present in ECJ including rich set of implemented metaheuristic algorithms, logging and statistics facilities, and distributed evaluation. The support of distributed evaluation is important for the search-based testing because evaluation of an individual takes a considerable time due to the need of multiple executions of the test. This way, we can easily distribute the evaluation among several computers and save time.

Export and Analysis of Results

The secondary aim of our infrastructure is to build a database of results which can be used to further study concurrency behaviour and effectiveness of noise injection techniques. The state space storage module allows to store results on a disk or in a database such that the results can be imported if needed. Each exported record contains a state identification and the obtained results including the execution time, error manifestation, and coverage metrics. This allows us to handle large sets of results which do not fit into memory. We also implemented an executor module plug-in which uses the stored results when asked to evaluate already evaluated state. This allows us to compare different search algorithms on the same sample of data and so mask the influence of scheduling non-determinism, and to accelerate evaluation because retrieving data from the disk or database is faster than the test execution.

The state space storage module also allows us to produce two dimensional tables containing various statistical data concerning the evaluated states of the state space. These data are suitable for further analysis and reasoning about the evaluation process. The data can be directly uploaded to the Universal Java Matrix Package library which allows us to visualise the results (e.g., to study the saturation effect), perform linear algebra operations, and to export the table into various formats including the Comma-separated values (CSV), Excel spreadsheet (XLS), and SQL database.

SearchBestie is able to produce a huge amount of statistical data in a form of views. Data mining techniques are ideal to deal with such data. Therefore, we used the RapidMiner tool to further analyse our data. So far, we only used the tool only to detect outliers which in most cases indicate an error in our infrastructure and to build complex database queries. Moreover, there is no problem to apply advanced data mining techniques to the collected data produced by our infrastructure. This is part of our future work.