
**BRNO UNIVERSITY OF TECHNOLOGY**
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

**FACULTY OF INFORMATION TECHNOLOGY**
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

**DEPARTMENT OF INTELLIGENT SYSTEMS**
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ


# APPLICATION OF GENETIC ALGORITHMS AND DATA MINING IN NOISE-BASED TESTING OF CONCURRENT SOFTWARE

**VYUŽITÍ TECHNIK GENETICKÝCH ALGORITMŮ A DOLOVÁNÍ Z DAT V TESTOVÁNÍ PARALELNÍCH PROGRAMŮ S VYUŽITÍM VKLÁDÁNÍ ŠUMU**

**PHD THESIS**
DISERTAČNÍ PRÁCE


**AUTHOR** — **Mgr. Bc. HANA PLUHÁČKOVÁ**
AUTOR PRÁCE

**SUPERVISOR** — **Prof. Ing. TOMÁŠ VOJNAR, Ph.D.**
ŠKOLITEL



**BRNO 2019**

# Abstract

This thesis proposes an improvement of the efficiency of testing concurrent software by employing data mining techniques and genetic algorithms in the process of testing concurrent software. Concurrent, or multi-threaded, programming has become very popular over the last few years. However, as the concurrent programming is far more demanding the sequential programming, its increased use leads to a significant increase in the number of errors that appear in commercial software due to errors in synchronization. Finding such errors using traditional testing methods is difficult. Moreover, repeated test executions of traditional testing that are performed in the same environment will typically examine similar interleavings only. Hence, the noise-based injection approach is used for influencing the scheduling by injecting various kinds of noise (delays, context switches, and so on) into the common thread behaviour which stress the software and can to show some rare behaviour. However, for the noise injection to be efficient, one has to choose suitable noise injection heuristics from among the many existing ones as well as to suitably choose values of their various parameters, which is not easy. In this work, there are used data mining methods and genetic algorithms and their combinations to deal with the problem of choosing such noise injection heuristics and values of their parameters. Besides setting up of the goals of the thesis, this proposal also provides a brief summary of the state of the art in application of data mining techniques and genetic algorithms to program testing problems.



# Abstrakt

Tato práce navrhuje zlepšení výkonu testování programů použitím technik dolování z dat a genetických algoritmů při testování paralelních programů. Paralelní programování se v posledních letech stává velmi populárním i přesto, že toto programování je mnohem náročnější než jednodušší sekvenční a proto jeho zvýšené používání vede k podstatně vyššímu počtu chyb. Tyto chyby se vyskytují v důsledku chyb v synchronizaci jednotlivých procesů programu. Nalezení takových chyb tradičním způsobem je složité a navíc opakované spouštění těchto testů ve stejném prostředí typicky vede pouze k prohledávání stejných prokládání. V práci se využívá metody vstřikování šumu, která vystresuje program tak, že se mohou objevit některá nová chování. Pro účinnost této metody je nutné zvolit vhodné heuristiky a též i hodnoty jejich parametrů, což není snadné. V práci se využívá metod dolování z dat, genetických algoritmů a jejich kombinace pro nalezení těchto heuristik a hodnot parametrů. V práci je vedle výsledků výzkumu uveden stručný přehled dalších technik testování paralelních programů.



# Keywords

testing, concurrent programs, data mining, genetic algorithms, AdaBoost, LASSO algorithm, noise injection

# Klíčová slova

testování, paralelní programy, dolování z dat, genetické algoritmy, AdaBoost, LASSO algoritmus, vstřikování šumu


# Reference

PLUHÁČKOVÁ, Hana. *Application of Genetic Algorithms and Data Mining in Noise-based Testing of Concurrent Software.* Brno, 2019. PhD thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Vojnar Tomáš.

# Application of Genetic Algorithms and Data Mining in Noise-based Testing of Concurrent Software

## Declaration

Hereby I declare that this PhD thesis was prepared as an original author's work under the supervision of Prof. Ing. Tomáš Vojnar, Ph.D. The supplementary information was provided by Dr. Zdeněk Letko. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

.....
Hana Pluháčková
September 30, 2019


## Acknowledgements

I would like express my gratitude to Prof. Ing. Tomáš Vojnar, Ph.D. for his supervision of my work, as well as to Dr. Zdeněk Letko for all the advice he provided. I would also like to thank all the people from the VeriFIT research group.

# Contents

# Chapter 1

# Introduction

Since programming is demanding and programmers always make mistakes, it is important to verify programs as carefully as possible. However, program verification is not easy, and some errors are very difficult to find. On the other hand, when a program fails, the consequences can be very expensive.

An example of such an expensive failure is the software failure that interrupted the New York Mercantile Exchange and telephone service to several East Coast cities in February 1998. Overall, estimates of the economic costs of faulty software in the U.S. range in tens of billions of dollars per year and they present approximately just under 1 percent of the nation's gross domestic product [96].

Hence, proper methods for finding errors in computer programs and/or for verifying their correctness are highly needed, and a lot of research effort is invested into developing new approaches for analysis, verification, and testing.

## 1.1 Analysis and Verification of Programs

There are various approaches how to analyze and verify programs and how to detect errors in the programs. From a high-level point of view, these methods can be divided to (1) methods of *testing and dynamic analysis*, and (2) methods of *static analysis* ranging from light-weight approaches (error patterns) to heavier-weight approaches (such as model checking, abstract interpretation, or theorem proving). Some of the latter approaches can be considered as formal verification approaches that can prove correctness of a system with respect to a specification (not just find errors).

An ideal verification tool would be a tool that has the following features: *full automation* (no human help is needed), *soundness* (a program found correct is indeed correct, i.e., no false negatives), *completeness* (reported errors are real; no false alarms), and *termination* (meaning that verification always terminates). However, due to undecidability and state explosion, the ideal is usually not achievable. Many verification methods do not guarantee termination and/or can cause false alarms, are not fully automatic, or do not scale well. In the following paragraphs, the basic types of analysis and verification methods are introduced in some more detail.

**Program Testing**. In program testing, a programmer writes a test or the test is generated from a high-level specification. An error in the program or in the test case is detected if the expected output is not achieved or if the program fails before producing

the output. Program testing checks the code along the execution trace of the test case only. This method is the most common way of finding errors in programs nowadays.

**Dynamic Analysis**. This technique also detects errors along execution traces. However, instead of checking outputs of a test, dynamic analysis automatically gathers information about the execution (the order of locking, the order of accessing shared memory locations, etc.) and analyses the gathered information with an intention to discover abnormal execution conditions. Usually, some kind of instrumentation that injects some additional code into the original code is used to gather the information. The information can be analyzed *on-the-fly*, during the execution, or *post-mortem*, after the end of the execution. Although the analysis gathers information concerning a single or several executions, sometimes, if some approximation is performed, it can discover even errors that are not directly on the witnessed execution traces. In the best-case scenario, a dynamic analysis is sound and complete with respect to the examined execution traces, but it is usually unsound with respect to all possible execution traces.

**Static Analysis**. Static analysis is based on a *compile-time* analysis. Some static analyses require for the code to be compilable only, although some heavy-weight static analysis approaches need the code to be runnable, too. These methods usually infer abstraction of the program behaviour from the code and try to find errors in this abstraction. Due to the over-approximation used, the methods often suffer from false positives. The code coverage may be total; sometimes static analysis even analyzes *dead code* that is never used along any possible execution trace (this is also a source of incompleteness) [89]. Static analysis includes various techniques, such as *model checking*, which is an example of the heavy-weight approaches that need a runnable code, *theorem proving*, a deductive verification method, often similar to the traditional mathematical theorem proving beginning with axioms, or *abstract interpretation*, a general approach that evaluates the program over suitable abstract domains, ignoring some details of the concrete semantics.

## 1.2 Verification of Concurrent Software

Concurrent programs belong among those where there is a very high chance of programmers making mistakes but which are also very difficult to verify. These programs have often very large state space due to many possible interleavings of the threads, and errors often hide in some rare, corner-case interleavings that involve some tricky interplay of the threads that the programmers did not think of.

Heavier-weight formal methods of verification, such as model checking [19], aim at precise program verification. Unfortunately, these precise approaches do not scale well for complex concurrent software. This is one of the main reasons why heuristic approaches such as light-weight static analysis, testing, and dynamic analysis are very popular in this area. While light-weight static analysis may scale, it often produces many false alarms (or it must be heavily fine-tuned for the given verification scenario — often for the price of suppressing some real errors together with the false ones).

When dealing with concurrent programs, testing and dynamic analysis that rely on executing the system under test (SUT) and evaluating the witnessed run are complicated by having to deal with the non-deterministic scheduling of program threads. Due to this problem, a single execution of a program is insufficient to find errors in the program even for the particular input data used in the execution. Moreover, even if the program has

been executed many times with the given input without spotting any failure, it is still possible that its future execution with exactly the same input will produce an incorrect result. A problem is that repeated testing in the same environment usually does not explore schedules that are too different.

One approach that is commonly accepted as a way to significantly improve on the above problem is the so-called *noise injection* (other common approaches are mentioned in Section 2.1). The noise injection approach [40] is based on heuristically disturbing the scheduling of program threads in hope of observing scheduling scenarios unseen so far. Although this approach cannot prove correctness of a program even under some bounds on its behaviour, it was demonstrated in [40, 62, 22] that it can rapidly increase the probability of spotting concurrency errors without introducing any false alarms. The noise injection approach is described in more detail in Section 2.2.

## 1.3 Goals of Thesis

The thesis is focused on concurrent software testing based on noise injection. As we have already sketched above and as we will discuss in more detail later on, this type of testing can stress programs in such a way that there manifest uncommon behaviours and interleavings of threads. This can be used to reveal rare errors that are otherwise extremely difficult to find. On the other hand, noise injection has many parameters that need to be suitably set (together with parameters of the programs under test themselves), and finding the right setting is difficult.

The main goal of the thesis is hence to improve the efficiency of the current methods of testing concurrent programs using noise injection by simplifying the process of finding the right settings of noise and test parameters. In the work, various approaches for finding suitable values of parameters of tests and noise are studied. In particular, those include data mining techniques, genetic algorithms and their combination, as well as further heuristics, such as exploitation of dependencies among testing under metrics of different cost.

## 1.4 Plan of Thesis and Overview of Achieved Results

The rest of the thesis is organized as described below. Chapter 2 introduces major underlying concepts and methods, on which the presented research builds, namely noise-based testing, selected concurrency metrics, basics of mathematical methods including data mining and basics of optimization approaches (such as genetic algorithms). This chapter also presents related works that are not specific for the individual parts of the thesis, the tools which we used for analyzing concurrent programs, and we also present the multi-threaded benchmarks which are used in the experiments.

The other chapters present the contribution of the thesis and are organized chronologically wrt. our publication results.

Chapter 3 introduces our methods based on the *multi-objective genetic algorithm* (*MOGA*) that we proposed for setting test and noise parameters of noise-based injection. This approach is compared with the older approach proposed for the same purpose, namely, setting of test and noise parameters by means of a *single-objective genetic algorithm* (*SOGA*). This approach was proposed within the VeriFIT research group before the beginning of the work on this thesis. Our research focused on using MOGA for setting test and noise parameters, including its comparison with the SOGA approach, was published as a conference

paper at the SSBSE'14 conference [44], and as the technical report [43] in cooperation with the ORT Braude College in Karmiel, Israel within a Kontakt II project.

The main goal of Chapter 4 is to use data mining to resolve the test and noise configuration problem. For this purpose, the AdaBoost approach, which is subsequently modified for better results in the given area, is suggested for use. The second goal of the chapter is a combination of the AdaBoost approach with genetic algorithms. This combination shows that both methods have their advantages. The AdaBoost approach was presented as the conference paper [6] at the MEMICS'14 conference. The modification of the AdaBoost approach was presented as a student poster at the AERFAI/INIT 2015 Summer School on Machine Learning in Benicassim and the results were also published as the journal paper [7] in the journal of Concurrency and Computation: Practice and Experience. We were also invited to the European Conference about Data Analysis (ECDA'18), where the combination of the AdaBoost approach and genetic algorithms was presented.

Chapter 5 is focused on improving the time needed for noise-based testing. In particular, for measuring the results of the test, there exist some concurrency metrics. Testing under some of them is more time-consuming but the metrics provide more information. On the other hand, testing under some metrics is less time-consuming, but they give less information. The different costs manifest, of course, during finding of the right parameters of the SUT and of the noise generation too. Hence the main idea of our next result is to try to identify dependencies between parameter settings suitable for testing under metrics of different costs and then use testing under a cheaper metric to find settings suitable for a more expensive metric. Alternatively, testing under several cheaper metrics can be used for this purpose too. This idea was presented at the EUROCAST'17 conference and published as the conference paper [64]. The next goal of this chapter is to find the optimal number of cheaper metrics for prediction of the given metrics. For this purpose, three approaches — using two, three, and four cheap metrics for the prediction — are compared. In the chapter there is also discussed a combination of the prediction approach with genetic algorithms.

Finally, Chapter 6 presents a summary of the results, concludes this PhD thesis, and introduces possible future research directions. Some preliminary results obtained within one of these directions were presented as a poster at the students' poster session during the conference MEMICS'17 (and unfortunately, not further developed due to a loss of the collaborating MSc student).

# Chapter 2

# Preliminaries

In this chapter, we introduce the basics of several areas which form the basis of the following chapters—namely: Section 2.1 presents an overview of different approaches to testing and analysis of concurrent programs. A more detailed introduction to noise based testing is provided in Section 2.2. Section 2.3 presents different types of metrics that can be used to measure coverage of behaviour of concurrent programs achieved during testing. An overview of the mathematical methods used for analyzing and verification of the concurrent programs is in Section 2.4, focusing on data mining and genetic algorithms that we later use. In Section 2.5 there is presented the tool support used for our experiments. Finally, Section 2.6 introduces multi-threaded benchmarks used as our case studies.

## 2.1 Testing and Analysis of Multi-threaded Programs

In the following section, there are presented overview of different testing methods which are used for testing and analyzing of the concurrent programs. Namely, there are presented examples of using stress, noise injection, systematic testing, dynamic analysis, coverage-driven testing and active testing.

Simple *stress testing* which is based on execution of a large number of threads competing for shared resources has been shown to increase the possibility of spotting concurrency errors only a little [83]. It has been also discussed many times that only small number of threads (usually two) are sufficient to detect concurrency errors and that concurrency errors manifest themselves only under specific interleaving scenario(s), e.g. [78, 107, 24].

The *noise injection* technique [24, 94] influences thread interleavings by inserting small delays, called *noise*, into the execution of selected threads. If there is another enabled thread, the noise cause switch of threads without much hurt to the performance of the application. The noise is inserted at random or based on specific parametrized heuristics which targets specific classes of concurrency errors. Many different noise heuristics can be used for this purpose [62]. The efficiency of the approach depends on the nature of the system under test (SUT) and the testing environment, which includes the way noise is generated [62]. A proper choice of *noise seeding heuristics* (e.g. calling `sleep` or `yield` statements, halting selected threads, etc.), *noise placement heuristics* (purely random, at selected statements, etc.), as well as of the values of the many parameters of these heuristics (such as strength, frequency, etc.) can rapidly increase the probability of detecting an error, but on the other hand, improper noise injection can hide it [58]. A proper selection of the noise heuristics

and their parameters is not easy, and it is often done by random [44]. More details about noise heuristics and parameters is in Section 2.2.

Among the main alternatives to noise-based testing, we first mention the so-called *systematic testing* [41, 9, 104, 71, 46, 45, 107]. The main idea of systematic testing is to control the scheduling of threads and systematically enumerates their different interleavings. Unlike noise-based testing, systematic testing provides better guarantees that a concurrency-related error will be found if present, and it can avoid re-execution of the same schedules. On the other hand, despite many heuristic optimizations that have been proposed, due to a need to systematically enumerate different schedules, systematic testing is still more heavy-weight than noise-based testing. Moreover, systematic testing can have problems with programs containing sources of non-determinism such as user input, external client requests, etc.

The systematic testing approach can be also seen as execution-based model checking which systematically tests as many thread interleaving scenarios as possible. The number of possible interleavings is often huge and therefore these techniques works with abstract and/or considerably bounded models of the SUT. The technique is therefore suitable mainly for unit testing in which is the technique able to discover and test all possible interleavings (with respect to the used abstraction and/or bounds). The technique is also suitable for debugging because the same recorded interleaving scenario can be enforced in the next execution of the test. Disadvantages of the technique include performance degradation due to need for dynamic computing and storing of the considered SUT thread interleavings model. The technique also suffer from problems with other sources of non-determinism in SUT, for instance, non-determinism caused by i/o operations.

Testing of concurrent programs can be combined with *dynamic analysis* [26, 52] which collects various pieces of information along the executed path and tries to detect errors which are in the SUT but did not necessarily occur during the execution. Many problem-specific dynamic analyses have been proposed for detecting special classes of errors, such as data races [26], atomicity violations [72], or deadlocks [11]. Most of the analyses are unsound and therefore can sometimes produce false alarms. Efficiency of dynamic analysis can be increased when a different execution path is analyzed during each execution of the test. A combination of noise injection or deterministic testing and dynamic analysis can thus lead to a synergy effect [22].

*Coverage-driven testing* as proposed in [107] and implemented in the Maple tool attempts to influence the scheduling such that the obtained coverage of several important synchronization idioms (called iRoots) is maximized. These idioms capture several important memory access patterns that are shown to be often related with error occurrences. Maple uses several heuristics to likely increase the coverage of iRoots. The technique provides lower guarantees of finding an error than systematic testing, but it is more scalable. The approach of Maple does not support some kinds of bugs (e.g. value-dependent bugs or some forms of deadlocks). Interestingly, multiple of the heuristics it uses are based on randomization. Maple can thus be viewed as being in between of systematic testing and noise-based testing (note that some of our noise placement heuristics are based on maximizing coverage too). An interesting question for future work is thus whether an approach for finding suitable values of noise parameters, such as the one we propose in this thesis, could be combined with the heuristics used in Maple too.

Finally, various combinations of the above approaches have been studied in the literature. In *active testing*, which is considered, e.g. in [90, 82, 53], some bug detector based on static analysis or extrapolating dynamic analysis is used to detect possible concurrency

errors and then some form of noise-based testing, directed by information from the first phase, is used to check whether the detected error is real. In [29], an approach combining noise-based testing and extrapolating dynamic analysis in the first phase was combined with bounded software model checking along the (partially) recorded trace from the first phase and in its neighbourhood. Such a combined approach can benefit from the techniques presented in this thesis too.

## 2.2 Noise Injection Techniques

As we have already said, noise injection disturbs the common scheduling of concurrently executing threads in order to allow for testing less common (but legal) schedules. In figure 2.1, we illustrate two of the possible effects that noise injection can have. Figure 2.1(a) illustrates a scenario in that the usual order in which two threads execute some events is swapped by noise injection (e.g. by an inserted delay). This can uncover a bug that happens only if the events happen in the swapped order. Note that if the swapped order can happen with noise injection, then the programmer did not exclude it using any synchronization means, and it can happen even without noise injection. If there was some synchronization in place, noise injection could not overcome it. This is, no new behaviour is introduced; just without noise injection, the probability of the events happening in the swapped order may be very low. Figure 2.1(b) then shows a situation where noise injection prolongs the time spend by a thread in a critical section, which can lead to another thread executing its critical section in parallel with the first one, possibly causing some concurrency error. As before, if such an error happens, it is a real error since the programmer did not prevent the situation by using any synchronization means, which noise injection would not be able to overcome. Thus, the situation can happen even without noise injection, though perhaps with a much lower probability.

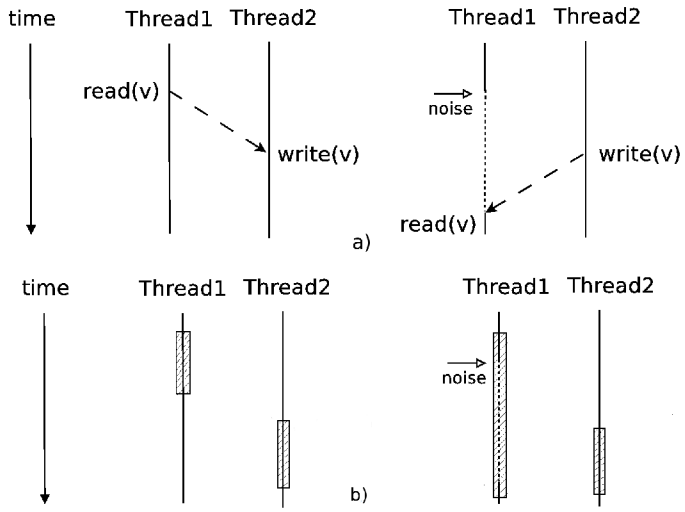



Figure 2.1: Two examples of the effect of noise injection: (a) reordering of the common order of two events in a concurrent program execution and (b) prolongation of the time spent by a thread in a critical section, leading to an overlapped execution of two critical sections.

We now provide some more technical details on noise injection. A thorough discussion of the technique can be found, e.g. in [30]. Noise injection heavily depends on two kinds of heuristics—namely, *noise seeding heuristics* and *noise placement heuristics*. The noise seeding heuristics determine the type and strength of the generated noise whereas the noise placement heuristics determine at what instants of program executions the noise gets injected.

### 2.2.1 Noise Seeding Heuristics

The basic noise seeding heuristics are: *yield*, *sleep*, *wait*, *busyWait*, *synchYield*, and *mixed*. The *yield* and *sleep* heuristics inject calls of the `yield()` and `sleep()` methods, respectively. In the case of the *wait* heuristic, the concerned threads must first obtain a special shared monitor, then call the `wait()` method, and finally release the monitor. The *synchYield* heuristic combines the *yield* heuristic with obtaining the monitor as in the case of the *wait* heuristic. The *busyWait* heuristic inserts a busy-waiting loop that is executed for some time. Finally, the *mixed* heuristic randomly chooses one of the five other basic heuristics at each noise injection location.

The additional noise seeding heuristics are: *haltOneThread* and *timeoutTamper*. The *haltOneThread* technique occasionally stops one thread until any other thread cannot run. The *timeoutTamper* heuristic randomly reduces the time-outs used when calling `sleep()` in the tested program (to test that programmers do not try to synchronize their threads by explicitly delaying some events).

All the above mentioned seeding techniques are parametrized by the so-called *strength of noise*. In the case of the *sleep* and *wait* heuristics, the strength gives the time to wait. In the case of the *yield* heuristic, the strength says how many times the `yield()` method should be called.

### 2.2.2 Noise Placement Heuristics

The noise placement heuristics are: the *random* heuristic, the *sharedVarNoise* heuristic, and the *coverage-based* heuristic. The *random* heuristic injects noise with some probability before every concurrency-related event in the program execution. The *sharedVarNoise* heuristic allows one to focus noise primarily at accesses to shared variables. There are two versions of this heuristic: *sharedVarNoise-all* which targets all accesses to shared variables and *sharedVarNoise-one* which targets accesses to a single randomly chosen shared variable in each test execution. Moreover, for both of these heuristics, one can decide whether the noise should be inserted solely when accessing shared variables or also at synchronisation operations such as locking (the so-called *nonVariableNoise* heuristic).

The *coverage-based* heuristic is based on collecting information about pairs of subsequent accesses to a shared variable from different threads and on inserting noise before further executions of the program instruction by which the given variable was accessed first (or before acquiring the shared lock that guards the given access provided there is such a lock). This is motivated by trying to reverse the ordering in which threads access variables.

As we have mentioned already above, the noise placement heuristics inject noise at the selected points of program executions with some probability. This probability is determined by the *noise frequency* parameter. The values of this parameter range from never inserting a noise to always inserting it. Additionally, the *coverage-based* heuristic can be extended by another heuristic (denoted as the *coverage-based-frequency* heuristic) that monitors the frequency with which a program location is visited during testing and

injects noise at the given program location with a probability adjusted according to this frequency—the more often a program location is executed the lower probability is used.

### 2.2.3 Test and Noise Configuration Search Problem

The *test and noise configuration search problem* (the TNCS problem) is formulated 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 parametrized 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. Analogically, 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*.

## 2.3 Measuring Quality of Testing Multi-threaded Programs

An important role in modern testing is played by the metrics which measure how well the SUT has been tested. This functionality is often provided by *coverage metrics* which measure how many of considered goals (based on selected coverage criteria) have been targeted by the tests. Coverage metrics which handle thread interleavings precisely [73] are hard to enumerate statically and effectively use due to potentially huge number of possible interleavings. On the other side, coverage metrics which do not consider thread interleavings at all, such as *synchronization coverage* [12], are insufficient because achieving full coverage does not mean that the program cannot contain concurrency errors.

In [61], there are presented an alternative coverage metrics based on coverage criteria which considers internal states to which a selected dynamic analysis algorithm can get. Such metrics naturally abstract away all behaviour of the SUT which are not important in order to detect (or cause) particular type of concurrency error. Still, coverage goals of these metrics are hard to compute statically and therefore such metrics are suitable mainly in saturation-based [91] and search-based testing which will be introduced next. In these approaches, the coverage metrics are used mainly to compare different results or to observe evolution of the testing process which do not require to know what is the full coverage.

The deterministic testing approaches discussed above benefit from model of SUT they dynamically build and maintain. Usually the model has form of a graph and therefore graph coverage metrics can be used to measure how well SUT has been tested. Since the model is constructed dynamically and the approaches has no knowledge on how big the model could potentially be, the problem with determination of full coverage remains for them as well.

In the paragraphs below, there are presented in detail the concurrency metrics which are used in this thesis within the experiments. All descriptions are taken from the paper [30].

**ConcurPairs.** The ConcurPairs coverage is based on concurrently executing instructions. It 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*. Results *false* means that the two locations are visited by the same thread. On the other hand, the *true* means that there occurred a context switch between the two program locations. This metric provides statement coverage information — using *false* mark – and interleaving information when it is used *true* mark at once. A task of this metric is denoted as a tuple $(pl_1, pl_2, switch)$ below. Variables $pl_1, pl_2 \in P$ represent here the consecutive program locations (only concurrency primitives and variable accesses are monitored), and $switch \in \{true, false\}$ indicates whether the context switch occurs in between of them.

**DUPairs.** It is a definition-use coverage which is based on the *all-du-path* coverage metric from parallel programs. The metric considers coverage tasks in the form of triples $(var, n_u^i, n_v^j)$ where $n_u^i$ is the $u^{th}$ node in the thread $T_i$ where the value of program variable *var* is defined while it is referenced in $v^{th}$ node in the thread $T_j$. A path in a Parallel Program Flow Graph (PPFG) covers such coverage task if the value of variable *var* is the first defined by thread $T_i$ and then the same value is used in $T_j$. This can be only guaranteed if a synchronization among threads $T_i$ quite simple model of parallel computation, for instance, it supports *post* and *wait* system of synchronization and *thread-create* operation for creating new threads only, just the master thread is allowed to create worker threads, and the number of created threads in a program need to be determined statically. Under this limitation, it is possible to number the particular threads. When dealing with today real-life applications, one cannot apply such restrictions. The original coverage metric was therefore slightly modified. The modified metric is referenced to as DUPairs* below. The coverage tasks of this metric has the form of tuples $(var, pl_1, pl_2, t_1, t_2)$ meaning that value of variable *var* is defined at program location $pl_1$ in the thread $t_1$ and then used at program location $pl_2$ in the thread $t_2$. Instead of precise numbering of individual threads the metric uses an abstract thread identification.

**Synchronization Coverage (Synchro).** The synchronization coverage focuses on the use of synchronization 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 synchronization primitives. For instance, in the case of a synchronized block (defined using the Java keyword *synchronized*), the obtained tasks are: *synchronization visited, synchronization blocking,* and *synchronization blocked.* The synchronization visited task is basically just a code coverage task. The other two are reported when there is an actual contention between synchronized blocks—when a thread $t_1$ reaches a synchronized block $A$ and stops because another thread $t_2$ is inside a block $B$ synchronized on the same lock. In this case, $A$ is reported as blocked, and $B$ as blocking (both, in addition, as visited). Tasks of this metric are denoted as tuples of the form $(pl_1, mode)$ where $pl_1 \in P$ represents the program location of a synchronization primitive, and *mode* represents an element from the set of the distinctive situations relevant for the given type of synchronization.

**Coverage Metrics Based on Avio.** The Avio algorithm detects atomicity violation over one variable and 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 $B$. Serializability 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 $var$. 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 unserializable execution. Tracking of all accesses that occur concurrently to a block $B$ can be very expensive. Therefore, a criterion to consider only the last interleaving access to the concerned variable from a different thread is defined. The basic Avio metric uses coverage tasks in the form of tuples $(pl_1, pl_2, pl_3, var)$ where the considered atomic block $B$ spans between program locations $pl_1 \in P$ and $pl_2 \in P$ where the variable $var \in V$ is accessed by a thread $t_1 \in T$ while it interferes with the access from a different thread $t_2 \in T, t_2 \neq t_1$ at program location $pl_3 \in P$. The extended metric Avio* incorporates into coverage tasks also information about the threads from which the accesses have been made resulting in tuples of the form $(pl_1, pl_2, pl_3, var, t_1, t_2)$. Single threaded programs cannot generate any such coverage task because basic as well as extended version of Avio-based coverage metric requires the variable $var$ to be accessed by two distinct threads.

**Coverage Metrics Based on Eraser.** The coverage metrics Eraser and Eraser* are based on the Eraser algorithm. For each thread, the algorithm computes a set of locks currently held by the thread, and for each variable access, the algorithm uses these sets to derive the set of locks that were held by each thread that had so far accessed the variable. These so-called locksets are maintained according to a *state* assigned to each variable which represents how the variable has been operated so far (e.g. exclusively within one thread, shared among threads, for reading only, etc.). Basic coverage tasks have the form of a tuple $(pl, var, state, lockset)$ where $pl \in P$ identifies the program location of an instruction accessing a shared variable $var \in V$, $state \in \{virgin, exclusive, exclusive', shared, modified, race\}$ indicates the state in which the Eraser's finite control automaton is when the given location is reached (the extended version of Eraser using the *exclusive'* state is considered), and $lockset \subseteq L$ denotes a set of locks currently guarding the variable $var$. Eraser* extends the basic Eraser metric by identification of a thread $t \in T$ performing the access operation. Extended coverage tasks thus have the form of $(pl, var, state, lockset, t)$. Accessing a variable $var$ at a certain program location $pl$ is a code coverage task which is here enriched by the information whether the variable has been already initialized (indicated by $virgin$ or $exclusive$ state). Other possible values of the state cannot be reached in single threaded applications.

**Coverage Metrics Based on GoldiLocks.** GoldiLocks is an advanced lockset-based algorithm which combines the use of locksets with computing the happens-before relation that says which events are *guaranteed* to happen before other events. In GoldiLocks, locksets are allowed to contain not only locks (L) but also variables (O) and threads (T). If a thread $t$ appears in the lockset of a variable when the variable is accessed, it means that $t$ is properly synchronized for using the given variable because all other accesses that might cause a data race are guaranteed to happen before the current access. The algorithm uses a limited number of elements placed in the lockset to represent an important part of the synchronization history preceding an access to a shared variable. The basic GoldiLocks algorithm is still relatively expensive but can be optimized by the so-called short circuit checks (SC) which are three cheap checks that are sufficient for deciding race freedom between the two last accesses to a variable. The original algorithm is then used only when SC cannot prove race freedom. The basic GoldiLock metric is based on coverage tasks

having the form of tuples $(pl, var, goldiLockSet)$ where $pl \in P$ gives the location of an instruction accessing a variable $var \in V$ and $goldiLockSet \subseteq O \cup L \cup T$ represents the lockset computed by GoldiLock. The tuple can be extended by a thread $t \in T$ which accesses the variable $var$ getting GoldiLock* coverage tasks of the form $(pl, var, goldiLockSet, t)$. Program location $pl$ at which the variable $var$ has been accessed represents a code coverage task. For single threaded applications, one of the short circuit checks discovers that data race cannot occur and the information about execution history captured in *goldiLockSet* can thus only distinguish the first access to the variable from the others.

**Coverage Metrics Based on GoodLock.** GoodLock is a popular deadlock detection algorithm that has several implementations' the metric presented here builds on the implementation published by Bensalem and Havelund. The algorithm builds the so-called *guarded lock graph* which is a labeled 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. The metrics focus on occurrence of nested locking that is considered interesting by GoodLock. Collection of the locksets of the threads which the original algorithm uses as one element of the edge label is omitted because this information is used in the algorithm to suppress certain false alarms only. The GoodLock metric is therefore based on coverage tasks in the form of tuples $(pl_1, pl_2, l_1, l_2)$ meaning that some thread $t \in T$ has first obtained the lock $l_1 \in L$ at the location $pl_1 \in P$ and later requested the lock $l_2 \in L$ at the location $pl_2 \in P$. The extended metric GoodLock* incorporates also identification of the thread $t$ forming the tuple $(pl_1, pl_2, l_1, l_2, t)$. Locks are usually used for synchronization of accesses to shared resources among several threads, however, also a single threaded application can request for locks and thus generate GoodLock-based coverage tasks.

**Coverage Metrics Based on Happens-Before Pairs.** These coverage metrics are motivated by observations obtained from the GoldiLocks algorithm and the vector-clock algorithms, both of them depend on computation of the happens-before relation. In order to get rid of the 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, the metrics focus on pieces of information the algorithms use for creating their representations of the analyzed program behaviours. All of these algorithms rely on synchronization events observed along the execution path. Inspired by this, the metrics capture successful synchronization events based on locks, volatile variables, wait-notify operations, and thread start and join operations used in Java. A basic coverage task is defined as a tuple $(pl_1, pl_2, syncObj)$ where $pl_1 \in P$ is a program location in a thread $t_1 \in T$ that was synchronized with the location $pl_2 \in P$ of the thread $t_2 \in T, t_2 \neq t_1$ using the synchronization object *syncObj*. The extended metric HBPair* incorporates identification of the synchronized threads forming the task as a tuple $(pl_1, pl_2, syncObj, t_1, t_2)$. In the same way as for the Avio-based metrics, no single threaded application can generate any HBPair or HBPair* coverage task because it captures a synchronization between two distinct threads only.

**Datarace.** The *Datarace* metric measures the number of warnings issued by the chosen data race detector. In our case, we use the GoldiLock algorithm for this purpose. Thus, metric says how many times the algorithm was successful and reported a possible error.

## 2.4 Methods Used in Thesis

This section provides an introduction to the areas of the methods that were used in the approaches proposed in the thesis and during the evaluation of the experiments. It covers main statistical approaches, data mining, and genetic algorithms.

Statistical methods are mainly used in the evaluation of tests, e.g. to compare the results of different approaches by the Student's t-value (statistical hypothesis about whether two approaches are significantly different or not) or standard numerical characteristics, such as average, variation, median or standard deviation. These methods were used in the experimental parts of the thesis for an evaluation of the tests we performed.

Approaches used in this work are introduced in more detail in the following two sections where basic data mining methods are presented and base of genetic algorithms.

### 2.4.1 Basic Data Mining Algorithms

Data mining allows us to answer a number of problems in different ways. There are four basic methods in data mining: (1) classification, (2) regression, (3) association rules, and (4) clustering [66]. Only two of them, namely, classification and regression, are introduced, as those have been used in the methods proposed in this work.

Both of these methods are so-called methods with a supervisor. Supervised learning uses predictive models that have a matrix $\mathbf{X}$ as their input and a vector $y$ as their output. The input matrix represents features, i.e. attributes of the given data sets. The output vector could be represented by categorical values (i.e. categories such as TRUE vs FALSE, or A, B, C, D, E and F as the classification in school) which is the case of classification or pattern recognition, or could be represented by real values, which is the case of regression. In both cases, a data set is divided into the training and the validation sample. The prediction model is created on the training sample and then it is tested on the validation sample. The validation sample is used for evaluating accuracy of the created model.

**Classification**. As mentioned in the previous text, the classification task consists of assigning variables from a given data set, described by a set of discrete- or continuous-valued attributes, to a set of classes, which can be considered values of a selected discrete target attribute. There are two main methods of classification: binary and multiclass. Classification approaches include decision trees, boosted trees, Naïev Bayes, and K-Nearest Neighbours.

For our purposes, the test and noise parameters are marked as variables and we want to assign a specific combination of the variables to the one of the two possible classes depending on the given goal of program testing. Here, the classes mean whether the given setting of the test has a higher probability of meeting the given goal. In Chapter 4, an approach based on boosted decision trees called AdaBoost is used for classifying of program testing.

**Regression**. The regression task consists of assignment of a numerical value to variables from a given data set, described by a set of discrete- or continuous-valued attributes. This assignment is supposed to approximate some target function, generally unknown, except for a subset of the data set – training sample. This training sample can be used to create the regression model that makes prediction of unknown target function values for any possible variable from the same data set feasible. In practical applications, the target function represents an interesting property of variables from the data set that either is

difficult and costly to determine, or (more typically) becomes known later than is needed. Among the regression approaches, there are linear regression or regression trees.

For our purposes, we have variables such as coverage metrics, where we count the number of tasks visible during the test execution of concurrent programs and our goal is to increase the coverage. In Chapter 5, three different regression algorithms are compared that could combine more metrics for prediction of the coverage of the metrics which are more time-consuming to collect.

As mentioned above, the data set is divided on training and validation samples in the supervised methods. The precision of the obtained classifier/model should be evaluated on the validation set. Notions of *accuracy* and *sensitivity*, based on the following quantities [57], can be used for that purpose:

- The number $TP$ of *true positives* that is the number of correctly classified positive examples, i.e. those objects $\overline{x}$ where $(\overline{x}, 1) \in \mathcal{V}$ and $F(\overline{x}) = 1$.
- The number $FP$ of *false positives* that is the number of wrongly classified negative examples, i.e. those objects $\overline{x}$ where $(\overline{x}, -1) \in \mathcal{V}$ but $F(\overline{x}) = 1$.
- The number $TN$ of *true negatives* that is the number of correctly classified negative examples, i.e. those objects $\overline{x}$ where $(\overline{x}, -1) \in \mathcal{V}$ and $F(\overline{x}) = -1$.
- The number $FN$ of *false negatives* that is the number of wrongly classified negative examples, i.e. those objects $\overline{x}$ where $(\overline{x}, 1) \in \mathcal{V}$ but $F(\overline{x}) = -1$.

Accuracy then gives the probability of a successful classification and can be computed as the fraction of the number of correctly classified items and the total number of items:

$$accuracy = \frac{TP + TN}{TP + FP + TN + FN}.$$

On the other hand, sensitivity (also called the true positive rate or TPR) expresses the fraction of correctly classified positive results and can be computed as the number of the items that were correctly classified positively divided by the sum of the correctly positively and incorrectly negatively classified items (for example, see [103]):

$$sensitivity = \frac{TP}{TP + FN}.$$

### 2.4.2 Genetic Algorithms

In this section, we briefly introduce genetic and evolutionary algorithms. More detailed information is presented in 3.2.1. These algorithms generally produce high-quality models. On the downside, they are very time-consuming. The following paragraphs introduce the basics of the genetic algorithms that will be used as an optimization method in the process of noise-based testing and dynamic analysis of concurrent programs.

The evolutionary algorithm (EA) tries to find the best solution possible from a search space of candidate solutions with respect to selected criteria. EA is suitable for problems with a huge search space, for which finding the best solution by the brute force approach is not feasible. In the context of EA, candidate solutions are called *individuals* and the set of all candidates solution is referred to as *individual space*. The individual space is mapped into the set of parameters associated with candidate solutions that is called *decision space*.

The specific values of parameters from this decision space for particular individuals are called *decision vector*. The decision vector corresponds to a genome in biology and a single parameter from the vector corresponds to a gene. Individuals are evaluated by *objective functions* resulting in an *objective vector* of specific values for particular objectives. Each such objective is related to a criterion applied on a candidate solution. The evaluation of the objective can be based on a single gene, however, it can be influenced by the whole genome as well. To compare candidate solutions in order to determine which of them is the best one, so-called *fitness function* combining the evaluation of all desired criteria into a single number is needed [108]. On the other hand, there is also another possibility where the fitness function focuses on evaluation of the given desired criteria separately — this type of fitness function is discussed in detail in Chapter 3.

A rather successful meta-heuristic search technique for complex optimization problems is the *genetic algorithm* (GA) [97], which is inspired by the process of natural selection. GA tries to find the best solutions by biased sampling of the solution search space, starting with an initial set (called a *generation*) of candidate solutions (also referred to as *individuals*). Each individual in the current population is evaluated and assigned a value called *fitness*, representing the suitability of the particular solution. The next generation of individuals is obtained from the current generation, typically by using stochastic recombination (called a *crossover*) of individuals selected according to their fitness and *mutation* of the new individual's attributes (called *genes*) in order for the search to not get stuck in the local extreme.

**Search Process of GA.** A subset of an individual space with a constant size is called a *population*. GA starts with an initial population and evaluates all its members (i.e. candidate solutions) by a fitness function. Based on this evaluation, the fitting individuals called *parents* are chosen by *selection operators* to generate new individuals called *children*. New individuals are usually the result of a crossover of two parents followed by a mutation. This process, called *breeding*, proceeds until a new child population is completed. New generations are gradually created until a sufficiently good solution is found or the maximum number of generations is created.

**Selection Operators.** Parents from the current population can be chosen for breeding using different techniques. *Fitness-Proportionate Selection* selects individuals proportionally to their fitness—individuals with higher fitness have higher probability to be selected for breeding than individuals with lower fitness [75]. *Tournament Selection* is based on a tournament. A specific number of individuals is randomly selected from the current population and the one with the highest fitness is taken for breeding [75]. For multi-objective optimization, *Mating Scheme* may be considered as a selection technique. Mating Scheme is slightly more complicated as it works in two phases and selects both parents. Within the first phase, a certain number of individuals is randomly selected for the first parent (group A) and the same number of individuals for the second parent (group B). In the second phase, the best individual from group A is selected for breeding while the individual from group B that is most similar to the parent from group A is selected for breeding [47].

**Crossover.** When two parents are selected for breeding, crossover takes place—two new individuals are created by a recombination of genomes of the parents (i.e. by exchanging parts of their decision vectors). The most common crossover techniques are *One-Point*,

*Two-Point*, and *Uniform Crossover* [75]. When the One-Point crossover is applied, the crossing occurs at one place only. The place of crossing $c$ is chosen between 1 and the length of the genome $l$. New individuals are obtained by exchanging genes of parents from place $c$ to the end of their genomes. For the Two-Point crossover, two places of crossing $c_1$ and $c_2$ are chosen, both between 1 and $l$, and new individuals are obtained by exchanging genes of the parents just between places $c_1$ and $c_2$. The Uniform Crossover technique goes through the whole genomes and exchanges pairs of corresponding genes with the preset probability.

**Mutation.** Mutation is applied on a single individual—each gene from the individual's genome is with a preset probability replaced by any value permissible for this gene.

## 2.5 Tool Support Used in Thesis

This section provides a description of the tools that were used for our experiments with testing of concurrent programs. The main tool is SearchBestie, which cooperates with IBM ConTest, and ECJ Toolkit [68].

### 2.5.1 SearchBestie

SearchBestie is a generic tool designated for solving search or optimization problems in the form of finding a combination of input parameters of a given system such that suits the tested system as well as the predefined goals of the testing. SearchBestie is in particular fine-tuned for the case when the system of interest is a concurrent program to be tested. The properties of interest can then be defined in two ways. The first approach is finding an error or warning by a dynamic analyser. However, since findings errors (in particular, rarely manifesting concurrency errors) is difficult, another target property can be the achieved coverage under some concurrency metric [61].

The name SearchBestie is an acronym for Search-Based Testing Enviroment. The goal of SearchBestie is not to execute the tests, but to resolve testing as a search problem. Execution and instrumentation of the tested Java program[1] is provided by external software that integrates into SearchBestie as a plug-in. An example of such software is ConTest that is introduced in the following subsection. The development of SearchBestie itself has been carried out in cooperation with researchers from IBM Haifa.

The architecture of SearchBestie consists of four cooperating modules: *Manager, State space storage, Search* and *Executor*. A general overview of the structure and functioning of the SearchBestie architecture is provided in Figure 2.2. The manager reads a configuration file and initializes 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 checking whether the 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 search is finished, the manager can analyze the obtained results or export them.

[1] SearchBestie is only created to test the programs written in Java language.

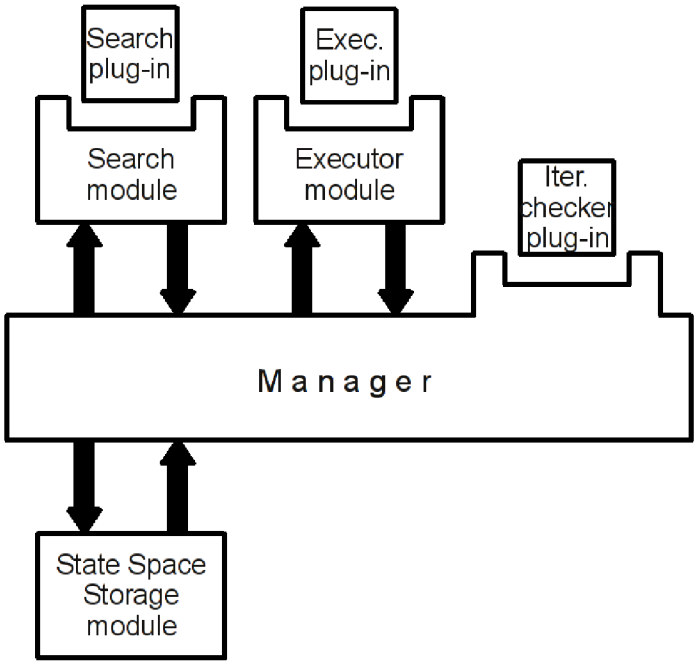



Figure 2.2: High-level architecture of SearchBestie [68].

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 the architecture is also supported by the idea of building blocks that allow for combining several plug-ins into more complicated entities.

### 2.5.2 IBM ConTest

The IBM Concurrency Testing tool (ConTest) has been developed and supported by researchers from the IBM Verification and Technologies group in Haifa, Israel. ConTest is an advanced testing solution and its main use is to expose and eliminate concurrency-related bugs in multi-threaded java applications[2][23].

A typical scenario of the ConTest use is that it performs instrumentation of Java byte-code before its execution first. When the instrumented code is executed, ConTest is initialized before executing the code of the test. During the initialization, ConTest reads its configuration files that contain a parameter setting 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. The data generated from the execution are available in sub-directories of the output directory after execution. Each generated file contains the ConTest unique execution identifier in its name. Finally, unnecessary data produced by ConTest or already processed data can be deleted.

[2]https://www.research.ibm.com/haifa/projects/verification/contest/

The cooperation of SearchBestie with ConTest is implemented within executor module plug-ins. The only activity required by the SearchBestie user is to enable testing with ConTest and properly set the parameters controlling its behaviour, the code to be executed, and parameters of ConTest and its plug-ins. The main functionality is implemented in the test engine that is responsible for generation of configuration files of ConTest and its plug-ins, execution of the test within a separate process and import of generated data from the ConTest output directory into the vector of results used by SearchBestie. 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 for detection of exceptions occurrence by observing outputs of the executed test. Processing the executed test outputs also allows for detection of 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.

### 2.5.3 ECJ toolkit

ECJ is a Java-based evolutionary computation system that has been 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 optimization, and differential evolution [102]. Its internal design allows one to easily interconnect SearchBestie with ECJ.

External tools like ECJ uses SearchBestie as a procedure for evaluation of candidate solutions. The cooperation works as follows. ECJ is executed by the user and within the ECJ initialization phase, SearchBestie is also initialized. ECJ then generates individuals for evaluation and performs a search. Each time ECJ requires an individual to be evaluated, SearchBestie is called. The evaluation consists of three steps: the individual is transformed into the corresponding state in the state space used by SearchBestie. Then the manager module evaluates the state as if it came from the search module. In the end 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 a predefined number of generations is evaluated, or by SeachBestie.

## 2.6 Case Studies

We now present the multi-threaded programs that are used as test cases in the experiments presented in the rest of the thesis.

***Airlines.*** The size of the test case is *0.3 kLOC*, *8 classes*. It is a small test case containing an *atomicity violation* error. It simulates an airline reservation system with three parameters $X$, $Y$, and $Z$: The system creates a flight whose capacity is $Z$ (number of available seats). Then, $X$ seller threads are executed, and they are periodically trying to get a seat on the flight. The parameter $Y$ controls how many iterations of an idle loop are done (and hence how much time is spent) between two successive attempts to book a ticket.

***Animator.*** The size of the test case is *1.5 kLOC*, *31 classes*. It is a program containing a *data race* and an *atomicity violation*. Animator is our short name for the XTANGO animation program [94] which is a general-purpose system for algorithm animation that allows programmers to create colourful, real-time, 2 & 1/2 dimensional, smooth animations

of their algorithms and programs. The focus of the system is on ease of use—programmers using this system need not be graphics experts to develop their own animations.

***Crawler.*** The size of this test case is *1.2 kLOC*, *19 classes*. The test case includes an *atomicity violation*. The program is taken from an IBM repository. It represents a skeleton of an IBM crawler product with a test environment simulating real usage of the system. Namely, the system creates a given number of threads waiting for a connection. If a connection is established, a worker thread serves it. Afterwards, when a given global time limit occurs, a shutdown sequence is initiated. This means that the working threads are not accepting new tasks, and, after finishing the current task, they die [59]. The bug present in the program manifests itself during the shutdown sequence but very rarely (roughly 15 times per 10,000 runs).

***Elevator.*** The size of this test case is *1.2 kLOC*, *12 classes*. The program contains a *data race* and an *atomicity violation*. It implements a real-time discrete-event simulation. The application is used as an example in a course on concurrent programming. Elevators are modeled as individual threads that poll directives from a central control board. Communication through the control board is synchronized through locks. The configuration used for our experiments simulates four elevators [84]. This benchmark has one parameter which controls the number of threads used.

***Rover.*** The size of the test case is *5.4 kLOC*, *82 classes*. Rover contains an *atomicity violation* and a *deadlock*. The K9 Rover from NASA Ames is an experimental platform for autonomous wheeled vehicles for exploration of a planetary surface such as Mars. The rover executive software prototype monitors executions of actions and performs responses and cleanup when the execution fails. In the configuration used in our experiments, eight threads are launched in the system [81]. This benchmark has one parameter which selects one of the available test scenarios.

***Cache4j.*** The size of this test case is *1.7 kLOC*, *66 classes*. Cache4j does not contain any known error. It is an LRU (Least Recently Used) lock-based cache implementation. The implementation is based on two internal data structures, a tree and a hash-map. The tree manages the LRU while the hash-map holds the data. The implementation is based on a single global lock [54].

***HEDC.*** The size of the test case is *12.7 kLOC*, *747 classes*. The program does not contain any known error. It represents an application kernel that implements a meta-crawler for searching multiple Internet archives in parallel. In our benchmark configuration, four principal threads issue random queries to two archives each. The individual queries are handled by a short random sleep interval of 0-200 ms; this ensures that the principal threads work out of sync. The application employs a library for concurrent programming by Doug Lea—in particular, the Pooled-Executor pattern. The workload and memory access pattern of this application kernel are typical for Internet server applications and similar to applications based on alternative mechanisms such as Java Servlets [84, 87].

***Moldyn.*** The size of the test case is *0.8 kLOC*, *14 classes*. It does not contain any known error. MolDyn is an N-body code modelling particles interacting under a Lennard-Jones potential in a cubic spatial volume with periodic boundary conditions. Performance is reported in interactions per second. The number of particles is given by N. The original

Fortran 77 code was written by Dieter Heerman, Institut für Theoretische Physik, Germany and converted to Java by Lorna Smith, EPCC.

***MonteCarlo.*** The size of the test case is *1.4 kLOC*, *22 classes*. It does not contain any known error. MonteCarlo is a financial simulation, using Monte Carlo techniques to price products derived from the price of an underlying asset. The code generates multiple time series with the same mean and fluctuation as a series of historical data. This benchmark has one parameter which controls the number of threads used for the computation [92].

***Raytracer.*** The size of the test case is *1.0 kLOC*, *22 classes*. It is without any known error. This benchmark measures the performance of a 3D ray tracer. The rendered scene contains 64 spheres, and it is rendered with a resolution of $N \times N$ pixels. The outermost loop (over rows of pixels) has been parallelised using a cyclic distribution for load balancing. This benchmark has one parameter controlling the number of threads used for the computation [80, 92].

***SOR.*** The size of the test case is *7.2 kLOC*, *256 classes*. The program does not contain any known error. SOR (Successive Over-Relaxation over a 2D grid) synchronizes its threads using a barrier rather than locks. It implements an iterative method for solving discretized Laplace equations on a grid data structure. In particular, it performs multiple passes over a rectangular grid until the values in the grid change less than a certain threshold, or a predefined number of iterations has been reached. The new value of a grid point is computed using a stencil operation, which depends only on the previous value of the point itself and its four neighbours in the grid. The program has two parameters: the number of iterations and the number of threads [80, 84].

***TSP.*** The size of this test case is *0.4 kLOC*, *8 classes*. It is without any known error. TSP (Travelling Salesman Problem) is a travelling salesman application which computes the shortest path for a salesperson to visit all cities in a given set exactly once, starting in one specific city. The program is parallelized by distributing the search space over different processors. Because the algorithm performs pruning, the amount of computation needed for each subspace is not known in advance and varies between different parts of the search space. Therefore, dynamic load balancing between the processors is needed. This benchmark has two parameters: the number of threads and a given input file with a TSP instance [80, 85].

# Chapter 3

# Application of Genetic Algorithms in Noise-based Testing

In this chapter, we propose an application of a multi-objective genetic algorithm to solve the TNCS problem. Within the proposal, we also suggest ways improve handling of with the inherent scheduling non-determinism in a genetic optimization process as well as of the much higher costs of some coverage tasks. Our work is motivated by a previous application of the single-objective genetic algorithm (SOGA) in the domain of noise-based testing of concurrent programs [42]. The SOGA approach improved the process of noise-based testing but came with significant problems on its own. As discussed in more details below, these problems concern the construction of a suitable fitness function aggregating all the objectives of interest in a way suitable for the highly non-deterministic environment of noise-based testing of concurrent programs. In particular with constructing a suitable fitness function aggregating all the objectives of interest.

In the following sections, we first discuss some specific related work and then provide the *multi-objective genetic algorithm* (MOGA). Afterwards, as our first contribution, we focus on selecting suitable objective functions that can be used within a multi-objective fitness function when solving the TNCS problem. This is needed since according to our experience, very significantly influence the quality of the search process. We particularly focus on the number of distinct values that the objectives can have, their correlation, and their tendency to suffer from the influence of non-determinism. Furthermore, we propose a novel modification of the coverage-based objective functions based on the so-called penalization of commonly achieved concurrency behaviour, which leads to quality improvement of the objectives wrt. the number of distinct values they can get and which guides the search process towards testing uncommon but legal behaviours.

Next, we compare the three commonly used multi-objective algorithms, namely, *SPEA*, *SPEA2*, and *NSGA-II*, with respect to their suitability for solving the TNCS problem. Subsequently, we study a suitable setting of parameters of the chosen algorithm to increase the quality of solutions discovered by this algorithm in our setting. Next, we present initial promising experiments demonstrating the ability of our MOGA approach to find good solutions of the TNCS problem and to suppress problems of the previous GA-based approach. Finally, we make a comparison of the previous GA-based approach with our new multi-objective genetic algorithm.

## 3.1 Related Work

This section provides an overview of works that apply metaheuristics and optimization techniques to the testing of multi-threaded programs.

Majority of the 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 (static) model checking* approach [37]. The basic idea of this approach is to explore areas of the state space that are more likely to contain concurrency errors first. Various algorithms, such as simulated annealing [15], genetic algorithms [37, 3], partial swarm optimisation (PSO) [15], ant colony optimisation (ACO) [2, 4], and estimation distribution algorithm (EDA) [93], were applied here.

The fitness functions used in these approaches are based on detection of error states (e.g. [93]), a distance to error manifestation (e.g. a high number of blocked threads can indicate that we are close to a deadlock [37, 3]) or formula-based heuristics [2, 4] which estimate the number of transitions required to get an objective node from the current one. Most of the approaches also search for a minimal counterexample path, i.e. a number of edges taken before the objective node is reached [93, 37, 3].

An advantage of this approach is that the underlying model checking offers a well-defined state space and a high degree of determinism. The disadvantage originates in the use of static model checkers, which do not scale well. Moreover, without exploring the entire state space, absence of errors cannot be proven. Therefore, we can consider such approaches as a heavy-weight deterministic testing.

Heuristic testing of multi-threaded programs using noise injection techniques is studied in [16] and [42]. In [16], the cross entropy heuristics is used to navigate the deterministic testing approach. Several fitness functions were proposed in this work for common non-concurrency errors, such as buffer overflow (a portion of buffer being used), and concurrency errors, such as a data race (a number of shared resources being accessed). In [42], a genetic algorithm was used to find a solution to the TNCS problem: the weighted fitness function combined detected errors, high concurrency related coverage and time.

Several other works focus on a slightly different problem of debugging multi-threaded programs, which tries to maximize the probability that a known error manifests during the test execution. In [28], genetic algorithms are applied to find a set of places in the program, where a noise should be placed to increase probability of spotting an error. In this case, the fitness function tries to minimize the number of places affected by the noise and favours solutions that put a high amount of noise to very small set of places.

The problem of increasing the probability of an error manifestation within the debugging process is targeted in [10, 99] as well. In [10], program locations are statically classified according to their suitability for the noise injection. Then a probabilistic algorithm is used to find a subset of program locations that increase the error manifestation ratio. In [99], a machine learning feature selection algorithm is used to identify a subset of program locations for noise injection. In this case, the test is executed many times. The program locations, to which the noise was injected in each execution, are collected together with information whether the error got manifested (or not).

A combination of the noise-based testing with GA appeared first in [42] (denoted as SOGA). There, a way of using the GA for finding a suitable setting of noise injection parameters was proposed and the following problems were identified. The combination via a *weighted fitness function* showed to be sensitive to the setting of weights. Finding weights that would be suitable in general turns out to be indeed very hard in the given

context, since different metrics can have very different ranges, which can moreover change from a test case to a test case. Apart from that, some of the metrics tend to correlate in some test cases, but not in others.

Furthermore, it was discovered that candidate solutions highly rated during one evaluation did not provide such good results when reevaluated again. This was caused by the thread scheduling non-determinism. It influenced the evaluation of candidate solutions too much despite the use of the accumulated evaluation over several test runs. It was also discovered that in some cases, the used genetic approach suffered from degradation, i.e. a quick loss of diversity in the population. It was caused by an excessive selection pressure on some objectives. Such a loss of diversity can unfortunately have a negative impact on the ability of the approach to achieve coverage tasks that are more difficult to cover, since they correspond to rare (and hence more likely to contain bugs unknown so far) program behaviours.

The aforementioned areas are what we aim to improve in this work. We also further improve the efficiency of solving the TNCS problem using the multi-objective optimization algorithms and novel fitness functions introduced in Section 3.3.

## 3.2 Preliminaries

Chapter 2.4.2 already contains general information about GA. In this preliminary section, we introduce more details about multi-objective optimizations, like the multi-objective evaluation of individuals. Then we introduce test cases and environment used for our experiments.

### 3.2.1 Multi-objective Genetic Algorithms

As mentioned above, individuals are evaluated by a fitness function, which represents the objective criteria of a problem to be solved by GA. The comparison of the individuals is quite easy if the evaluation is based on a single criterion only. Such a comparison is called a *Single-objective Optimization Problem (SOP)*. A more complicated situation occurs when more objective criteria need to be followed simultaneously—such a case is called a *Multi-objective Optimization Problem (MOP)* and is, actually, our case.

**The Single-objective Optimization.** In single-objective optimization, the set of candidate solutions needs to be completely (totally) ordered according to the fitness function $\mathbf{f}$, i.e. any two candidate solutions $\mathbf{a}, \mathbf{b} \in \mathbf{X}$ then either $\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{b})$ or $\mathbf{f}(\mathbf{a}) \leq \mathbf{f}(\mathbf{b})$ is true. The traditional approach to solve a multi-objective problem by the single-objective optimization is to bundle all objectives into a single scalar fitness function using a *weighted sum of objectives*

$$\mathbf{f}(\mathbf{x}) = w_1 * f_1(\mathbf{x}) + w_2 * f_2(\mathbf{x}) + \cdots + w_k * f_k(\mathbf{x}).$$

Using the weighted sum as the fitness function has several drawbacks. The obvious issue is how to set the weights $w_1, w_2, \ldots, w_k$ for particular objectives. The weights may reflect the importance of concrete objectives; however, they may also capture the balance between the objectives. A wrong setting of the weights can lead to neglecting some objectives. The other issue is non-linearity of objective values. Furthermore, it may not be possible to identify a single best solution for several multi-objective problems, because the individuals

are not totally ordered with respect to the given objectives—only a partial order can be found among them. For example, we want to buy an aircraft and we have two criteria: range ($f1$) and maximum cruise speed ($f2$). We want an aircraft with a high range and high cruise speed within the given budget, but these objectives go against each other. Figure 3.1 shows six different individuals. For instance, we can see that individual A has better cruise speed (i.e. higher $f2$) than individual B, who has a better range (i.e. higher $f1$) than individual A and thus, we cannot decide which aircraft is better. In such situations, it can be useful to examine the *Pareto dominance* or *Pareto non-dominance*.

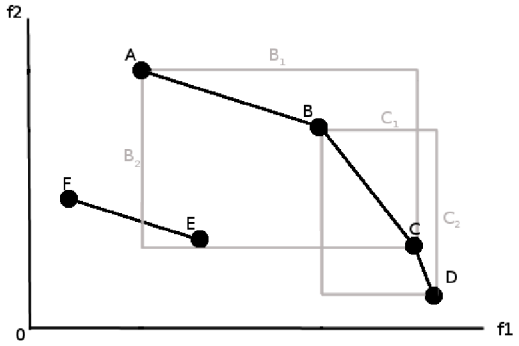



Figure 3.1: An example of Pareto ranks.

**The Multi-objective Optimization.** MOP in general consists of a set of $n$ parameters (i.e. decision variables), a set of $k$ objective functions, and a set of $m$ constraints on the decision variables [108]. The optimization goal is to maximize the objective vector $\mathbf{y}$:

$$\mathbf{y} = \mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_k(\mathbf{x}))$$

with respect to the constraints $\mathbf{e}$ delimiting the set of candidate solutions:

$$\mathbf{e}(\mathbf{x}) = (e_1(\mathbf{x}), e_2(\mathbf{x}), \ldots, e_m(\mathbf{x})) \leq \mathbf{0}$$

where $\mathbf{x} = (x_1, x_2, \ldots, x_n) \in \mathbf{X}$ is a decision vector from the decision space $\mathbf{X}$.

**Pareto Dominance.** For any two decision vectors $\mathbf{a}$ and $\mathbf{b}$ from the decision space $\mathbf{X}$,

$\mathbf{a} \succ \mathbf{b}$ (we say $\mathbf{a}$ dominates $\mathbf{b}$) *iff* $\mathbf{f}(\mathbf{a}) > \mathbf{f}(\mathbf{b})$

$\mathbf{a} \succeq \mathbf{b}$ (we say $\mathbf{a}$ weakly dominates $\mathbf{b}$) *iff* $\mathbf{f}(\mathbf{a}) \geq \mathbf{f}(\mathbf{b})$

$\mathbf{a} \sim \mathbf{b}$ (we say $\mathbf{a}$ is indifferent to $\mathbf{b}$) *iff* $\mathbf{f}(\mathbf{a}) \not\geq \mathbf{f}(\mathbf{b}) \wedge \mathbf{f}(\mathbf{b}) \not\geq \mathbf{f}(\mathbf{a})$

In our example (Figure 3.1), there are four individuals (A, B, C, and D) that are not dominated by any other individual.

**Pareto Optimality.** A decision vector $\mathbf{x} \in \mathbf{X}$ is said to be non-dominated regarding a set $\mathbf{A} \subseteq \mathbf{X}$ *iff* $\nexists \mathbf{a} \in \mathbf{A} : \mathbf{a} \succ \mathbf{x}$. The non-dominated decision vector $\mathbf{x}$ is called *Pareto optimal* and the set of all non-dominated decision vectors is called the ***Pareto-optimal front***.

In our example (Figure 3.1), the Pareto-optimal front or the *Pareto Front Rank* contains four individuals, namely, A, B, C, and D. If we then remove these individuals from our set, we can create the second Pareto front. It contains two individuals, namely, E and F, which are dominated by some individuals from the Pareto front, but they are not dominated by each other. The evaluation of one individual is not based on objective functions only, but it is also influenced by other individuals.

There are several algorithms for the multi-objective optimization that use different evaluation of individuals. However, all of them exploit the non-dominated sorting. For our purposes, we have analyzed the *Non-Dominated Sorting Genetic Algorithm II (NSGA-II)* and two versions of the *Strength Pareto Evolutionary Algorithm* (*SPEA* and *SPEA2)*.

**NSGA-II.** The skeleton of the NSGA-II is as follows: (1) we start with initial population $P$, (2) we compute the Pareto ranks of all individuals, (3) the best $n$ individuals are held in an *archive*, (4) we breed new population $Q$ from population $P$, (5) we compute the Pareto ranks of all individuals $P \cup Q$ and decide, which individuals stay in the archive, (6) new population $Q$ becomes population $P$, (7) the process continues with step (4) until we obtain the required solution or create the maximum number of generations.

**Sparsity.** To achieve better diversity among individuals from the same Pareto front, we can define sparsity. For instance, as the *Manhattan distance* over every objective between an individual's left and right neighbours [69]. The sparsity of outer individuals that have only one neighbour is defined as infinite. We illustrate the sparsity on individuals B and C from Figure 3.1. The sparsity of individual B (i.e. $|C_1 - A_1| + |C_2 - A_2|$) is higher than the sparsity of individual C (i.e. $|D_1 - B_1| + |D_2 - B_2|$).

**SPEA and SPEA2.** In SPEA, the value of fitness is not based directly on Pareto fronts, but on the so-called *strength*. The Pareto-optimal solutions found so far are stored in the archive, which is also referred to as an *external set*. The fitness of individuals from population $P$ is calculated using the strengths of individuals in the external set. At first, each individual $i$ from the external set $i \in \mathbf{ES}$ with decision vector $\mathbf{x}_i$ is assigned with strength $S(i)$ (a real value from $[0, 1)$). The strength represents the ratio between the number of individuals $j$ with decision vector $\mathbf{x}_j$, which are weakly dominated by individual $i$ and the size $N$ of population $P$ plus one.

$$S(i) = \frac{|\{j | j \in P \wedge \mathbf{x}_i \succeq \mathbf{x}_j\}|}{N + 1}$$

Fitness $F(i)$ of individual $i$ from the external set is equal to its strength. That is $F(i) = S(i)$ while fitness $F(j)$ of individual $j$ from population $P$ is equal to one plus the sum of strengths of individuals from the external set, which weakly dominate individual $j$.

$$F(j) = 1 + \sum_{i \in ES | \mathbf{x}_i \succeq \mathbf{x}_j} S(i)$$

Note that the value of such a fitness needs to be minimized here. The main steps of *SPEA* are the following, (1) initialization—initial population $P$, (2) updating—create or

update the external set, (3) fitness assignment—evaluation of individuals, (4) breeding—selection of parents, recombination, mutation, (5) termination—terminate search or go to step (2).

The weakness of the fitness evaluation within SPEA is the distribution of individuals in the external set. If the external set does not contain enough different individuals, the diversity of the evaluation is weak. Moreover, the diversity of evaluation is weak as well if the individuals in $P$ are close to each other, because such individuals are dominated by the same individuals from the external set and thus they have the same fitness value.

These problems are addressed by *SPEA2*, which evaluates individuals not only using the external set **ES**, but also using population $P$ itself. Strength $S(i)$ for each individual $i$ from $\mathrm{ES} \cup P$ represents the number of individuals that are dominated by $i$.

$$S(i) = |\{j | j \in ES \cup P \wedge \mathbf{x}_i \succ \mathbf{x}_j\}|$$

The raw fitness of individual $j$ is then calculated as the sum of strengths of its dominating individuals $i$. The fitness needs to be minimized here as well (actually, a zero value means a Pareto-optimal solution).

$$R(j) = \sum_{i \in ES \cup P | \mathbf{x}_i \succ \mathbf{x}_j} S(i)$$

For example, individual $j$ is dominated by individuals $i_1$ and $i_2$, individual $i_1$ dominates two individuals (i.e. $S(i_1) = 2$) and $i_2$ dominates three individuals (i.e. $S(i_2) = 3$). Then the raw fitness of individual $j$ is $R(j) = S(i_1) + S(i_2) = 5$.

Additional density information $D(i) \in (0, 1)$ is incorporated to discriminate between individuals having the same raw fitness value. The *SPEA2* uses $k$-th nearest neighbour method and adds the density to the raw fitness $F(i) = R(i) + D(i)$.

The main loop of the *SPEA2* algorithm is similar to *SPEA*: (1) initialization—initial population $P$, (2) fitness assignment—evaluation of all individuals from $P$, (3) environmental selection—fill the external set with $N$ best individuals from $ES \cup P$, (4) termination—if some terminating criteria is satisfied, (5) breeding—selection of a parents, recombination, mutation and continue with step (2).

*SPEA2* has a fixed size of the external set—compared to *SPEA*, where the size of external set depends on the size of the Pareto front rank (of course, if the number of Pareto-optimal solutions exceeds a predefined limit, some members are removed by a clustering technique to preserve the characteristics of the non-dominated front). *SPEA2* fills the size of the external set with dominated individuals [109, 110].

**Quantitative Traits and Realized Heritability.** As mentioned above, there are several similarities between the evaluation of individuals in genetic algorithms and in biology. Here, we would like to discuss another notion from biology that can be useful for our purpose. In biology, there are two main types of traits—qualitative and quantitative. The basic difference is that a qualitative trait is typically influenced by a single gen, while a quantitative trait is influenced by several gens and/or environment. This means that two genetically identical individuals can have different evaluation of the same trait. In our case, two individuals with the same decision vector can have different objective vectors and thus, we may use the evaluation inspired by quantitative traits.

*Heritability* indicates whether the variability of monitored traits is due to genetic factors, while *realized heritability* ($h^2$) is often used to quantify the degree to which a trait

in a population can be pushed by selection [17, 51]. We exploit heritability and realized heritability to confirm that our approach of using GA for the TNCS problem is useful. Realized heritability can be either calculated using several formulas or estimated statistically by linear regression. The regression coefficient reflects a relationship between the offspring evaluation and the parent evaluation. A low values $h^2$ (less than 0.01) occurs when the offspring of the selected parents differ a little from the original population. On the other hand, a high values of $h^2$ (more than 0.6) occurs when the offspring of the selected parents differ from the original population almost as much as the selected parents do. We use the realized heritability estimated by the linear regression technique to confirm the inheritability of selected objectives.

Linear regression is an approximation of given values by a line using a method of least squares. This line can be described using the following function:

$$y = \beta_0 + \beta_1 x + \epsilon$$

where $y$ is dependent variable—in our case, it represents values of offspring, $x$ is explanatory variable—in our case, it represents values of parents, $\beta_0$ and $\beta_1$ are regression parameters, and $\epsilon$ is noise. A vector of regression parameters $\beta$ can be estimated as follows:

$$\widehat{\beta} = \left(\mathbb{X}^T\mathbb{X}\right)^{-1}\mathbb{X}^T\mathbf{Y}$$

where $\mathbb{X}$ is a matrix with ones in the first column and the values of parents in the second column, and where $\mathbf{Y}$ is a vector of the values of offspring. If parameter $\beta_1$ (i.e. the slope of the regression line) is equal to 0, then there is no dependency between offspring and parents. The degree of the dependency, if indicated by the slope of the regression line, can be determined using either the coefficient of determination $r^2$ or correlation coefficient $r$. The coefficient of determination can be calculated as:

$$r^2 = 1 - \frac{S_e}{S_t}$$

where $S_e$ is the residual sum of squares and $S_t$ is the total sum of squares. These sums can be calculated as:

$$S_e = \left(\mathbf{Y} - \widehat{\mathbf{Y}}\right)^T \left(\mathbf{Y} - \widehat{\mathbf{Y}}\right)$$

where $\widehat{\mathbf{Y}} = \widehat{\beta}\mathbb{X}$ and

$$S_t = \left(\mathbf{Y} - \overline{\mathbf{Y}}\right)^T \left(\mathbf{Y} - \overline{\mathbf{Y}}\right)$$

where $\overline{\mathbf{Y}}$ is the arithmetic mean of $\mathbf{Y}$ components. Note that the coefficient of determination is square of the correlation coefficient. In our experiments, we have used the correlation coefficient for evaluation of dependency. It is from interval $\langle -1, 1\rangle$ and the value 0 means that there is no correlation between the values, i.e. offspring is independent of parents. Values of the correlation coefficient that are closer to $-1$ or 1 mean stronger dependency. The correlation coefficient is calculated using covariance between parents and their offspring divided by their standard deviations:

$$r = \frac{C\left(\mathbb{X}, \mathbf{Y}\right)}{s_{\mathbb{X}} s_{\mathbf{Y}}}$$

where $C\left(\mathbb{X}, \mathbf{Y}\right)$ is covariance calculated as

$$C\left(\mathbb{X}, \mathbf{Y}\right) = \sum \left(\mathbb{X} - \overline{\mathbb{X}}\right)\left(\mathbf{Y} - \overline{\mathbf{Y}}\right).$$

### 3.2.2 Test Cases and Environment We Use

In the following sections, we present various results obtained in our infrastructure for testing concurrent programs. This infrastructure is based on the SearchBestie [60] platform, which uses the IBM Concurrency Testing Tool (ConTest) [40] to inject noise into execution of considered programs and its plug-ins [59, 61] for dynamic analysis and collection of coverage metrics. The used meta-heuristic algorithms were implemented within the ECJ library [102], which cooperates with our SearchBestie platform as well.

Our initial observations presented below are based on data collected from the three multi-threaded Java programs, namely, Airlines, Animator, and Crawler. They were presented in Section 2.6 and each of them contains a concurrency error. Animator was executed on the Intel i7-3770K processor using Oracle JDK 1.6. The other two test cases were executed on a machine with two Intel X5355 processors using the same version of the Oracle JDK.

## 3.3 Multi-objective Genetic Solution of TNCS Problem

This section describes our proposal of applying MOGA to the TNCS problem. It touches upon several important aspects of setting MOGA for a successful application to the TNCS problem. In particular, we introduce possible objectives in Section 3.3.1 first. These can be used in construction of a successful fitness function, with emphasis on the selected properties that affect their suitability for our approach. Section 3.3.2 compares three popular multi-objective genetic algorithms (SPEA, SPEA2 and NSGA-II) with respect to their suitability for our approach. In Section 3.3.3, we discuss several aspects that influence selection of particular objectives and construction of suitable fitness functions for our MOGA approach. Finally, the setting of parameters of the particular MOGA (such as the size of population, mutation, and crossover operators) is discussed and experimentally evaluated in Section 3.3.4.

### 3.3.1 Important Properties of Considered Objectives

Various metrics can be collected from the execution of the instrumented programs. Our testing infrastructure is able to detect test failure, measure a duration of the test execution, and collect various code and concurrency coverage metrics as well as warnings produced by various attached dynamic analyzers, which are able to detect data races, atomicity violations, and deadlocks. Collecting all these data introduces a considerable slowdown. Moreover, some of the metrics are more suitable to be used as an objective and some are less suitable for this purpose. In this section, we discuss key properties of metrics suitable for the meta-heuristic approach, especially MOGA. We particularly focus on the number of distinctive values produced by metrics, correlation among objectives, and their stability. The stability here means the ability of the objective to provide similar values for the same decision vector in presence of the non-deterministic behaviour of tested multi-threaded programs.

**Number of Distinct Values Produced by Objectives.** One of the important properties of the considered objectives is their ability to classify the considered solutions. In general, many meta-heuristic algorithms provide worse results when the objectives with a low number of distinct values in objectives are used [97]. In our case, we indeed do have

metrics that suffer from the lack of distinct values. For instance, the *error* metrics provides us with a boolean value whether the test fails or passes. The number of distinct values can be slightly increased by multiple execution of the same test (this makes sense in presence of a non-deterministic behaviour of the executed multi-threaded programs). However, 10 executions gives us a possibility to classify the considered solution only into 10 groups according to the number of test executions that fail. A very small number of distinct values is also provided by metrics based on warnings produced by the dynamic analyses.

In [61], we discuss and compare a few coverage metrics from several perspectives, including the number of distinct values they provide. There are coverage metrics, such as *HBPair*, that focus only on the synchronization done among two threads and provide only a few distinct values for small programs or programs with only little synchronization. The more context information is included into the coverage metrics, the higher number of distinct values is usually obtained. For instance, the *Avio* metrics considers three subsequent accesses to a single shared variable from two threads. The *Avio*$^*$ metrics adds an identification determining from which two threads these accesses were performed and therefore again increases the number of distinct values. The *GoldiLockSC* metrics, which provides very good results in comparison, does not consider a direct identification of threads. Instead, the higher number of distinct values is obtained by considering the contents of a lockset produced by the GoldiLock algorithm [26] as context information.

High numbers of distinct values of an objective might be impractical in some cases. For instance, the *ConcurPair* coverage metrics [61] considers all the subsequent tuples of concurrency-related events. The concurrency aspect can be even emphasized by assigning different weights to the concurrency-related events executed by the same thread and the concurrency-related events executed in different threads (referenced as *WConcurPair* [42]). Handling and working with a huge number of coverage tasks of these metrics produced by a big, heavily concurrent program might be slow. However, for the small programs we use, the metrics provide fine-grained information about concurrency.

A very good candidate for a satisfactory objective from the point of view discussed here is time, because the length of the test execution can be measured in small units (e.g. milliseconds). However, time does not reflect concurrency and therefore is less attractive for us. On the other hand, it might be interesting to use it later on when searching for solutions that provide good results in a short time.

**Correlation of Objectives.** Another important property of the considered objectives is their correlation. If two objectives correlate, they contribute to the search with the same information. Therefore, it is recommended to use non-correlating objectives in meta-heuristic algorithms [97] so they do not need to bother with correlation themselves. As most of the considered concurrency coverage metrics focus on the concurrency behaviour, there is a high chance that some of them will correlate. Therefore, we analyzed all metrics proposed in [61] and used in [42], whether they correlate on our test cases. In particular, we performed 1000 executions of each of the considered test cases, namely, Animator, Airlines, and Crawler, with randomly chosen configurations of noise. During each execution, we collected all considered metrics and analyzed correlation among them.

Table 3.1 shows a fragment of our results focused only on metrics, which we mention in this section. Data for the correlation table are taken from all considered test cases. There is a high correlation (over 0.8) among the *Avio*$^*$, *GoldiLockSC*$^*$, *Eraser* and *DUPair* objectives, which focus on the same behaviour of the considered programs (i.e. the way how threads access shared variables). The *LockSet* metrics, which captures warnings pro-

duced by the lockset-based Eraser algorithm [88], naturally correlates with metrics based on locksets (i.e. *GoldiLockSC** and *Eraser*).

Objectives with low correlation with other shown objectives are *Time* and *Error*, which do not consider a concurrency-related behaviour at all. Rather small correlation with other concurrency-related metrics is provided by the *HBPair* and *WConcurPair* objectives, which focus on different behaviours. Specifically, *HBPair* considers synchronization among threads only and *WConcurPair* captures thread context switches.

Table 3.1: Correlation of objectives in all three considered test cases.

| | Time | Error | WConcur. | DUPair | HBPair | Avio* | GoldiSC* | Eraser | LockSet |
|---|---|---|---|---|---|---|---|---|---|
| Time | 1 | 0.140 | 0.343 | 0.506 | 0.164 | 0.344 | 0.481 | 0.507 | -0.373 |
| Error | 0.140 | 1 | -0.268 | -0.209 | -0.622 | -0.187 | -0.176 | -0.258 | 0.346 |
| WConcurPair | 0.343 | -0.268 | 1 | 0.746 | 0.658 | 0.900 | 0.743 | 0.765 | -0.571 |
| DUPair | 0.506 | -0.209 | 0.746 | 1 | 0.391 | 0.857 | 0.996 | 0.997 | -0.868 |
| HBPair | 0.164 | -0.622 | 0.658 | 0.391 | 1 | 0.557 | 0.342 | 0.459 | -0.288 |
| Avio* | 0.344 | -0.187 | 0.900 | 0.857 | 0.557 | 1 | 0.863 | 0.861 | -0.650 |
| GoldiLockSC* | 0.481 | -0.176 | 0.743 | 0.996 | 0.342 | 0.863 | 1 | 0.988 | -0.866 |
| Eraser | 0.507 | -0.258 | 0.765 | 0.997 | 0.459 | 0.861 | 0.988 | 1 | -0.863 |
| LockSet | -0.373 | 0.346 | -0.571 | -0.868 | -0.288 | -0.650 | -0.866 | -0.863 | 1 |

When studying the correlation tables created for particular test cases, we found that the correlation depends on the nature of the test case. For instance, in the Crawler test case, most of the objectives highly correlated—including the *HBPair* coverage, which does not correlate in the other considered test cases that much.

Overall, non-correlating objectives are *Time* and *Error*: they do correlate neither with each other nor with the concurrency-related coverage metrics. The coverage metrics correlation depends on the particular test case. In most cases, a lower correlation was detected among the *WConcurPair*, *HBPair* and *GoldiLockSC** metrics. As mentioned above, the most contributing factor in this phenomenon is the behaviour that is measured by these objectives.

**Stability of Objectives.** Another property of possible objectives, which we discuss here, is the ability to provide stable values in presence of non-determinism in execution of concurrent programs. The work on using GA to solve the TNCS problem [42] recognized that one of the major obstacles for applying GA in this domain is the non-deterministic behaviour of concurrent programs, which gets reflected in non-deterministic objective values. Specifically, if we run a single test with a single configuration multiple times, each run can give us different objective values. This can have a rather negative impact on our use of GA, since the same individual can be considered to give great results at some point during the breeding process and subsequently, it gives poor results only. Below, we briefly introduce several possibilities to increase stability when still considering just one representative value. Then we illustrate problems we faced and choose a suitable technique to reduce non-determinism.

The natural approach to reduce non-determinism is based on performing the experiment multiple times and use a suitable value (or values) to characterize the result (and, in some cases, a degree of non-determinism). In the work [42], the effects of non-determinism were reduced by using *cumulation* over test runs repeated several times (five in our case) with a single candidate solution (i.e. the configuration). However, this solution did not produce truly satisfactory results with respect to stability. Therefore, we now look into the possibility of using average, median, or modus values instead.

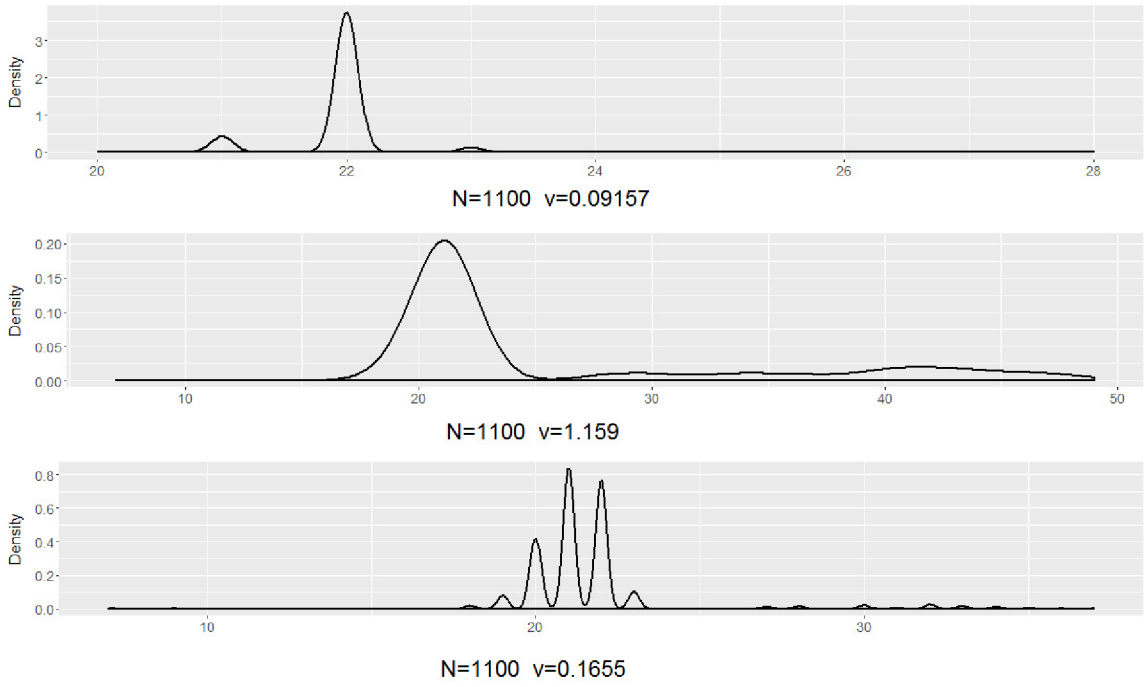



Figure 3.2: Illustration of three common distributions of values of a selected coverage metrics (*WConcurPair*) on Crawler test case. The first subgraph shows a normal distribution with a small variation, the second subgraph shows a distribution that tends to the normal one, but unfortunately with a high variation. The third subgraph shows a non-normal distribution.

The well-known statistical measures mentioned above are commonly used to characterize data sets. To choose between them, one has to understand what kind of data are to be characterized. Graphs in Figure 3.2 illustrate common outcomes of the considered objectives in a repeated execution of the same configuration. The figure shows three graphs representing the distribution of values of a selected objective (*WConcurPair* in this case) collected from 1100 independent executions (denoted $N$ in the figure) of three different test and noise configurations of the Crawler test case. The graphs were chosen to illustrate three main situations that represent common observations we got when analyzing different objectives on multiple executions of different test cases.

The first graph demonstrates an ideal situation when the distribution of observed values is normal with a few outliers. Moreover, the values in this case have a very small variation coefficient (denoted $v$ in the figure). In such cases, average, modus, and median values represent the data set quite accurately. Even the cumulated values, which we used previously, would characterize the observed data with a high stability in this case.

The second graph illustrates the most common situation when the distribution is *not normal* with several outliers and a high variation coefficient. In such cases, the average and the cumulation do not characterize the observed data accurately and therefore tend to be unstable. The modus and the median characterize such data sets with a better accuracy.

Finally, the third graph shows the very unpleasant situation, which we sometimes encounter as well. The variation coefficient is again high and the distribution is not normal,

but the number of outliers is very high too. In such cases, even the modus and the median do not characterize the results well. But, they are still more stable approaches than based on the average and cumulation values.

Below, we present extended comparison of the multiple benchmarks (eight concurrent programs, namely Airlines, Animator, Crawler, Elevator, MolDyn, MonteCarlo, Raytracer, and Rover—see Section 2.6) for three aforementioned approaches—median (*med*), mode (*mod*)[1], and the cumulative value (*cum*), which is computed as sum for time and as united coverage for the considered coverage metrics. For each of our case studies, we randomly selected 100 test configurations, executed each of them in 10 batches of 10 runs, and computed the representative values in several different ways for each batch. Afterwards, we compared stability of the representative values obtained across the batches. Table 3.2 shows the average values of variation coefficients of the representatives computed across all the considered configurations for each case study and each approach to/option of computing a representative value. It is clear that the best average stability was provided by the median.

Table 3.2: Stability of representatives.

| Case | med | mod | cum |
|---|---|---|---|
| Airlines | **0.033** | 0.054 | 0.051 |
| Animator | **0.012** | 0.027 | 0.092 |
| Crawler | **0.211** | 0.261 | 0.255 |
| Elevator | 0.145 | 0.227 | **0.107** |
| MolDyn | **0.020** | 0.025 | 0.024 |
| MonteCarlo | **0.015** | 0.019 | 0.022 |
| Raytracer | 0.022 | 0.020 | **0.016** |
| Rover | **0.059** | 0.100 | 0.141 |
| Average | **0.065** | 0.092 | 0.088 |

Based on these observations and information from literature [21], we decided to use modus (denoted *mod*) and median (denoted *med*) computed from metrics collected from multiple executions of the test and noise configuration. In particular, we decided to use modus for metrics that provide a small number of distinct values (e.g. errors) and median for the rest of metrics. In the future, we would like to use the variation coefficient in evaluation of configurations as well.

### 3.3.2 Selection of Multi-objective Genetic Algorithm

Another step needed to apply multi-objective genetic optimization to solve the TNCS problem is to choose a suitable multi-objective genetic algorithm. Therefore, in this subsection, we analyze the well-known multi-objective genetic algorithms *SPEA*, *SPEA2*, and *NSGA-II* introduced in Section 3.2.1 from the point of view of their applicability for solving the TNCS problem. We particularly concentrate on checking which of the evaluation functions $F_{spea}(i)$, $F_{spea2}(i)$, and $F_{nsga\text{-}II}(i)$ implemented by these algorithms provides the best results in classifying our individuals $i$ into a reasonable (i.e. neither too small nor too big) number of classes.

[1] Taking the biggest modus if there are several modus values.

To demonstrate differences among the algorithms, we chose to show the results that we obtained when experimenting with rather correlating objectives. In particular, we chose the Crawler test case and highly (coef. 0.966) correlated metrics *Avio*$^*$ and *GoldiLockSC*$^*$. We chose 40 different individuals $i$ (i.e. test and noise configurations) and evaluated each 11 times. We therefore obtained 440 different values, which should ideally be classified into 40 different fitness values by the considered algorithms. The correlation of objectives is not desirable, but different test cases behave differently. We therefore cannot rule out the correlated objectives completely. Our goal is to choose an algorithm that behaves well even under such circumstances.

Table 3.3: Problematic pairs of objectives and their evaluation by multi-objective fitness functions.

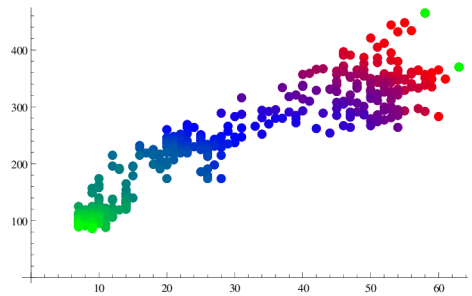
| Pair of objectives | *SPEA* | *SPEA2* | *NSGA-II* |
|---|---|---|---|
| (*Avio*$^*$, *GoldiLockSC*$^*$) | 4 | 366 | 106 |
| (*Time*, *GoldiLockSC*$^*$) | 30 | 410 | 38 |
| (*Time*, *Error*) | 7 | 437 | 386 |
| (*Error*, *GoldiLockSC*$^*$) | 8 | 240 | 199 |

**SPEA.** As mentioned in Section 3.2.1, evaluation of individuals by *SPEA* depends on the number and the distribution of individuals in the external set (the Pareto front rank). In our experiments, *SPEA* provided us with an insufficient number of different values of $F_{spea}(i)$ for different individuals $i$. In the experiment described above, we got four different fitness values only (the external set contains just two individuals in this case). *SPEA* does not provide us with a sufficient ability of classifying different individuals in this case.

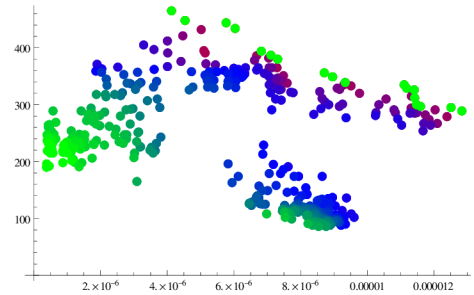
**SPEA2.** Compared with the *SPEA* algorithm, *SPEA2* improves the evaluation of individuals by taking into account not only the dominating individuals, but also the dominated ones. In our experiments, *SPEA2* had no more problems with a small number of individuals in the external set. For correlating objectives in our experiment, *SPEA2* achieved a much bigger number of different fitness values. On the other hand, our use of *SPEA2* led to another problem. Specifically, the number of the obtained fitness values got close to the number of the evaluated individuals (366 in this experiment), which is way too much.

**NSGA-II.** The last algorithm that we tried was *NSGA-II*. This algorithm finally gave us satisfactory results in that the number of the generated classes of individuals was neither too small nor too big. As explained in Section 3.2.1, *NSGA-II* assigns individuals into the so-called Pareto ranks that provide a basis for evaluation of individuals $i$ by $F_{nsga\text{-}II}(i)$. Subsequently, to achieve a better distribution of values along the Pareto front rank, the notion of sparsity is used. However, since sparsity concerns a single Pareto rank, we ignored it in the experiment presented here. *NSGA-II* produced 106 different fitness values in our experiment, which is a reasonable number for the given case.

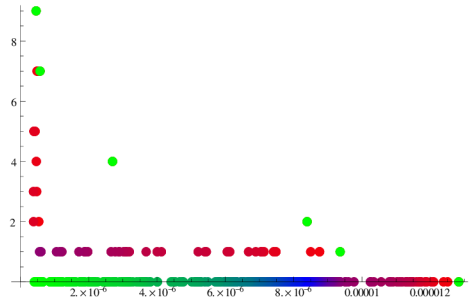
Table 3.3 shows a sample of our further experiments with the ability of the considered algorithms to provide satisfactory results when tuples of problematic objectives are used. In



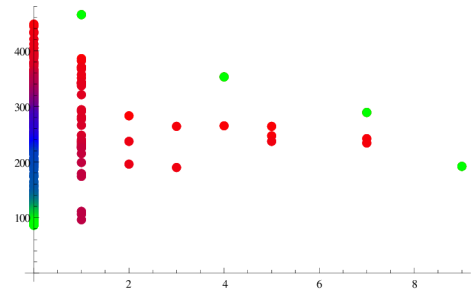
(a) $(Aviotript, Goldilocksct)$, 106 ranks



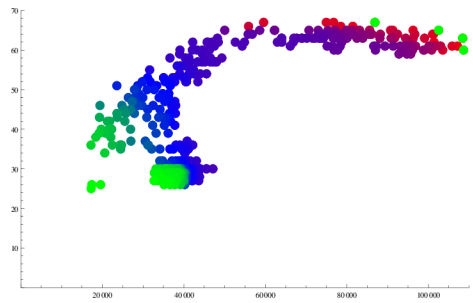
(b) $(testDuration, Goldilocksct)$, 38 ranks



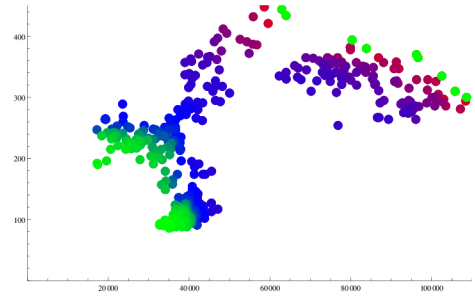
(c) $(testDuration, Error)$, 386 ranks



(d) $(Error, Goldilocksct)$, 199 ranks



(e) $(WConcurPairs, HBPair)$, 105 ranks



(f) $(WConcurPairs, Goldilocksct)$, 47 ranks

Figure 3.3: Graphs showing how *NSGA-II* handles different pairs of objectives.

particular, we focus on the problems of correlation (Avio* and GoldiLockSC*) and distinctive numbers of the objective (Error) that are too low. We chose the test duration (Time) objective to represent a good objective that does not correlate with others and provides a high number of distinctive values. The table shows the results for the same experiment as considered above (i.e. 40 individuals, 440 evaluations of the Crawler test case). The first row therefore summarizes the numbers discussed previously.

Table 3.3 also shows that *SPEA* produces too small numbers of fitness values, *SPEA2* too large numbers of values, which are close to the number of evaluations, whereas *NSGA-II* gives a reasonable value in between of the extremes (except the case of the *Time* and *Error* pair of objectives).

Figure 3.3 shows key points demonstrating why *NSGA-II* achieved the numbers shown in the table. The graphs show positions of the achieved values in the space (x-axis represents the first objective in a pair and y-axis represents the other objective) and different ranks computed by the algorithm (depicted using colours). The first four graphs (3.3(a)

to 3.3(d)) present situations considered in Table 3.3. Figures 3.3(e) and 3.3(f) show a situation when we chose good pairs of objectives. Specifically, we chose to present results for the (*WConcurPair*, *HBPair*) and (*WConcurPair*, *GoldiLockSC**) objectives, respectively.

All the major problems *NSGA-II* faces with our objectives are visible in the figures. Graphs 3.3(a) and 3.3(e) demonstrate the problem of correlating objectives, which the algorithm handles quite well. Graphs 3.3(c) and 3.3(d) emphasize the problem when one (Graph 3.3(d)) or both (Graph 3.3(c)) objectives provide an insufficient number of distinct values. Finally, Graphs 3.3(b) and 3.3(f) show an ideal situation when non-correlating objectives with a sufficient number of distinct values are used.

In the rest of our test cases, *NSGA-II* provided us with similarly good results too. Therefore, we consider *NSGA-II* to be the best algorithm out of the considered ones for our purposes. In the further analyses, we concentrate on its use only.

### 3.3.3 Selection of Objectives

In this section, we discuss several aspects of choosing objectives for our MOGA approach. First, we discuss the number of objectives considered by the chosen *NSGA-II* algorithm. Then, we discuss the final selection of objectives for the fitness function. Additionally, the way to emphasize achieving uncommon observations is presented. And finally, three fitness functions, which we later compare in Section 3.4, are proposed.

**Number of Considered Objectives.** As mentioned above, there are various metrics that can be used as objectives for our MOGA approach. The choice of the *NSGA-II* algorithm makes an important limitation to the number of considered objectives. It has been shown [48, 20] that the algorithm suffers from its ability to handle more than three objectives adequately. Therefore, we decided to choose the maximum, i.e. *three objectives*.

**Selection of Suitable Objectives.** Suitable objectives for our approach are those that have a high number of distinctive values and do not correlate. Moreover, the objectives should provide stable values in our inherently non-deterministic testing environment. They also need to reflect our goal: to achieve a high coverage of various concurrency behaviours and/or success in finding concurrency errors.

Based on the results presented in [61] and study of properties of the considered objectives summarized above, we chose three concurrency coverage metrics as candidates for good objectives: *HBPair*, *GoldiLockSC**, and *WConcurPair*. To minimize impact of non-determinism, we chose to consider as objective median of these coverage metrics, computed from five test executions with the same configuration. Five executions were chosen as a trade-off between a higher number of executions, which leads to a lower impact of non-determinism, and time-constraints, because each execution of SUT requires a considerable amount of time.

Since our goal is to find a concurrency error, we decided to also consider the *Error* and *LockSet* metrics, which provide information whether a concurrency error occurred during the execution and whether Eraser algorithm detected a problem in synchronization. We decided to use them in our experiments although they often provide an insufficient number of distinct values.

**Emphasize Uncommon Observations.** When analyzing the results of the tests, we noticed that some behaviour was observed every time we executed the test. Another be-

haviour was observed often (i.e. in more than 50% of executions) and some behaviour was rare. The goal of the testing is to observe not only the easily achieved behaviour, but also the behaviour that is hard to achieve. Therefore, we decided to *penalize* often (and therefore easily) achieved behaviours. The motivation behind this is to force the optimization algorithm to search for candidate solutions that can achieve a high coverage of behaviours that are not easily achievable. This should lead to a further improvement in the quality of our solutions.

Table 3.4: Impact of penalization on the number of distinctive values of coverage metrics.

| Test | Coverage | Penalization | Normal |
|---|---|---|---|
| Airlines | WConcurPairs | 95 | 87 |
| | HBPair | 1 | 1 |
| | GoldiLockSC* | 31 | 20 |
| Animator | WConcurPairs | 115 | 115 |
| | HBPair | 41 | 20 |
| | GoldiLockSC* | 115 | 110 |
| Crawler | WConcurPairs | 89 | 88 |
| | HBPair | 79 | 23 |
| | GoldiLockSC* | 86 | 57 |

The technical solution of the penalization works as follows. We let the genetic algorithm to evaluate the first generation (i.e. randomly chosen candidate solutions). Then we assign a probability to each covered task. The probability is assigned according to the number of executions it was observed in within the first generation. All the following test executions are evaluated with respect to these probabilities. This means that all behaviours not observed in the first generation add 1 to the considered value and the behaviours observed are penalized using the computed probability (if probability is 0.1, value 0.1 is taken).

This approach has also a positive side effect, i.e., the increase of the number of distinctive values our metrics can achieve. This observation is demonstrated in Table 3.4, which shows the number of distinctive values we achieved with and without penalization. The data in the table were collected from a randomly chosen MOGA experiment with the Crawler test case, population size 20, 100 generations and penalization enabled. The penalization was therefore computed from 20 candidate solutions of the initial population and applied to 1980 individuals from the following generations, from which the data are presented. The table clearly shows the increase of the number of distinctive values in case when the penalization is enabled.

**Selected Fitness Functions.** Considering the aforementioned findings, we identify the following fitness functions as potentially suitable for noise-based testing of concurrent software:

$fitness1a = (WConcurPair_{cum(5)}, HBPair_{cum(5)}, GoldiLockcSC^{*}_{cum(5)})$
$fitness1b = (WConcurPair_{med(5)}, HBPair_{med(5)}, GoldiLockcSC^{*}_{med(5)})$
$fitness2 = (Error_{mod(5)}, LockSet_{mod(5)}, GoldiLockcSC^{*}_{med(5)})$

The fitness functions differ in ways to increase stability of objectives. The *fitness1a* function uses cumulation, *fitness1b* and *fitness2* use median and for objectives with a low number of distinct values (i.e. *LockSet* and *Error*), modus is used. The *fitness2* function differs in considered objectives. It considers the number of concurrency errors (*Error*) and

the number of warnings produced by the Eraser algorithm (*LockSet*) combined with a selected concurrency coverage metrics (*GoldiLockSC**). The efficiency of the fitness functions is evaluated in Section 3.4.

### 3.3.4 Setting up Multi-objective Algorithm

Before the MOGA approach can be used effectively, a proper setting of its parameters is needed. Setting of the parameters such as size of population, number of generations, and selection, crossover, and mutation operators are presented here. Later, this step allows us to get most of the MOGA optimization and also to learn what makes MOGA successful. In this section, the previous experience obtained when solving the TNCS problem using GA [42] was used to choose the initial sets of suitable parameters of MOGA that are worth to experiment with.

In particular, we experiment with the population sizes and the number of generations in the way that the number of individual evaluations in one experiment remains constant (i.e. 2000 evaluations of individuals per experiment). Therefore, for populations of size 20, 40, and 100 we used sizes of 100, 50, and 20 generations, respectively. Furthermore, we studied the influence of three different crossover operators available in the ECJ toolkit [102] (called *one*, *two*, and *any*) and three different probabilities of mutation (0.01, 0.1, and 0.5). In total, we experimented with 27 different settings of MOGA (3 sizes of population, 3 crossover operators, and 3 mutation probabilities). The results presented below are based on the average values collected from multiple executions of each MOGA setting, which differ only in the initial random seed values (i.e. only in the individuals generated in the first generation).

The selection operator was the same in all the experiments. It was set to the Mating Scheme [47] selection algorithm, which provides better results for the NSGA algorithm [47] than the fitness-based tournament or the proportional selection algorithms that are commonly used in the single-objective GA. This algorithm combines fitness-based selection of one parent selected for crossover and the similarity-based tournament selection for the second parent. This algorithm also provided the best results in our preliminary experiments.

Hence our individuals are represented using vectors of integers (as discussed in Section 3.2), the crossover operators works as follows. The *one* crossover operator randomly splits two selected individuals into two parts and generates their offspring by random choosing between parents at each part of the vector. The *two* operator cuts the vector into three pieces of a random length and the *any* operator cuts the vector into elements.

The mutation operator, which we used randomly, selects an element of the vector and sets its value to a random value from the allowed range. All experiments were done only on the Crawler and the Airlines test cases introduced in Section 2.6, which represent test cases with reasonably short execution time (we used 324,000 executions of Crawler and 540,000 executions of the Airlines test case to collect data for the results presented here).

The MOGA approach was set to use only the *fitness1a* function, but the results were compared using multiple criteria considering the quality of the resulting individuals as well as the quality of the optimization process. Specifically, we computed (i) the variation of individuals in the last generation, (ii) the generation in which MOGA degenerated, (iii) the average achieved coverage obtained by the individuals from the last generation (we considered *WConcurPair*, *HBPair*, and *GoldiLockSC** metrics, which are used in *fitness1a*), and (iv) the accumulated number of detected errors in SUT during the experiment. The variation was computed as the number of different individuals in the last generation

divided by the population size. In our work, degeneration is considered a situation when at least half of the individuals in the population are identical.

The best configuration of MOGA was selected as follows: for each test case, we sorted the configurations according to each criterion described above and assigned numbers from 1 (the worst) to 27 (the best) to the individual configurations. Finally, we summed these numbers for all four considered criteria and test cases. Then, we chose the best configuration to be the one with the highest score achieved. Particularly, the best configuration consists of: the population size 20 (100 generations), the crossover type *two*, and the mutation probability 0.5 – and it is referred to as *MOGAconf* below. This configuration provides the best values in the variation, degeneration, and *HBPair* coverage metrics in both of the considered test cases and very good results in the other considered criteria.

Note that *MOGAconf* operates with a relatively high mutation probability (0.5), which helps in exploration of the new promising solutions. This is combined with a relatively high selection pressure. Our results show that this mixture helps MOGA to dramatically improve possible solutions during the first generations, because the incorporation of the *NSGA-II* archive helps to preserve the best solutions evaluated thus far in presence of such an agile search process.

## 3.4 General Experiments with MOGA Approach

This section contains the first of our experimental evaluation of the MOGA approach (not counting the preliminary experiments that we have presented in the previous section where they were used for properly setting various parameters of our MOGA approach). In particular, in Section 3.4.1, we first present experiments proving that our MOGA approach can indeed solve the TNCS optimization problem with a positive effect on the testing process despite the involved non-determinism. In these experiments, heritability, regression, and correlation introduced in Section 3.2 were used to show that the good individuals chosen by the MOGA indeed produce good offspring regardless of non-determinism present in the evaluation of individuals and that the MOGA approach can be effective in beating the random approach often used in the literature (or in practice). Subsequently, Section 3.4.2 contains a comparison of the fitness functions introduced in Section 3.3. The fitness functions are compared only in their ability to avoid degradation of the optimization process. In Section 3.5, we then proceed with experiments comparing our MOGA approach with the SOGA solution.

### 3.4.1 Ensuring that MOGA Works in Presence of Non-determinism

In this section, we present a set of experiments conducted to ensure the ability of our MOGA approach to actually improve the considered objectives regardless of non-determinism in evaluation of individuals. In these experiments, we applied MOGA with the *MOGAconf* setting on a set of three different test cases, namely, Airlines, Animator, and Crawler introduced in Section 2.6. We focus mainly on the *fitness1a* and *fitness1b* fitness functions proposed in Section 3.3.

To study the effect of non-determinism, we computed heritability, regression, and correlation to see whether there is a positive relation among parents and their offspring. In other words, we check whether parents selected by the selection operator indeed represent configurations that are able to steadily improve the testing performance with respect to considered objectives in presence of non-determinism in evaluation of individuals. Below,

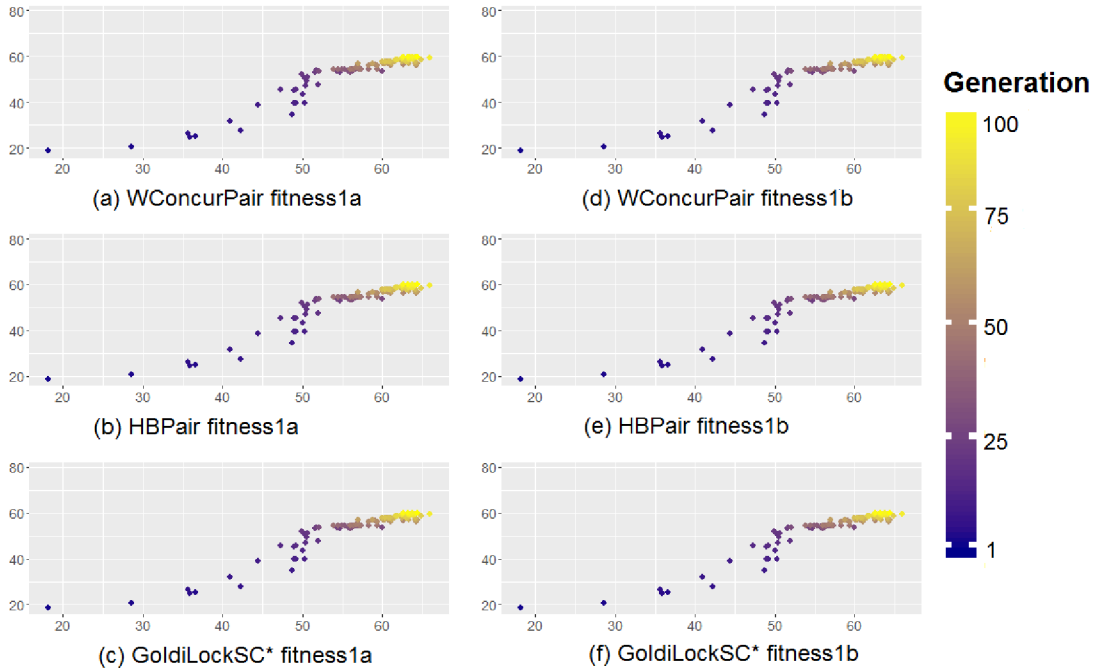


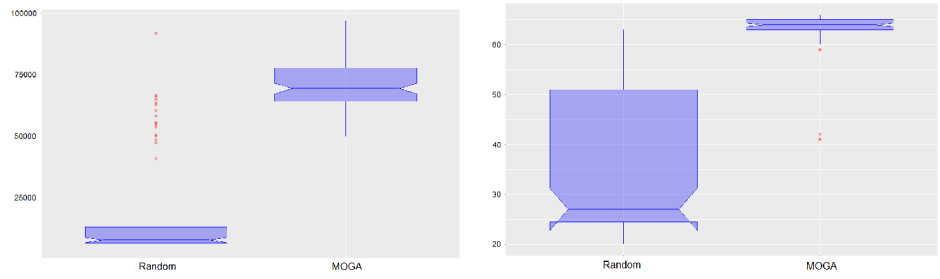

Figure 3.4: Regression graphs demonstrating ability of MOGA to improve achieved results in all three considered objectives.

we first show graphically on one test case the general tendency of MOGA to optimize solutions in presence of non-determinism. Then, we present on a study of correlation and heritability that the objectives (i.e. fitness function) must be carefully chosen to achieve such positive results.

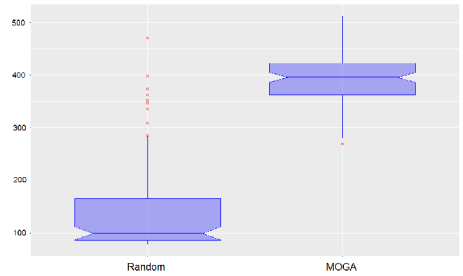
Graphs in Figure 3.4 illustrate a relation of particular objectives (namely, *WConcurPair*, *HBPair*, and *GoldiLockSC**) between parents (x axis) and their offspring (y axis) in the Crawler test case. Each point in the graph represents an average value of a particular objective (i.e. coverage) achieved by parents selected for breeding (x axis) and an average value of a objective achieved by offspring generated from these parents (y axis). The number of points in the graph therefore corresponds to the number of breedings. Moreover, the points are coloured to emphasizes the general tendency of MOGA to improve the available solutions. The dark blue points represent the first generations, the violet points are the next generations and so on to the yellow points, which represents the last generations.

Graphs 3.4 (a), 3.4 (b), and 3.4 (c) depict *fitness1a* and graphs 3.4 (d), 3.4 (e), and 3.4 (f) depict the same results when *fitness1b* was used. As described in Section 3.2, a positive slope emphasizes a high correlation among parents and their offspring, showing that good parents produce good offspring. Moreover, it is also clear that the initial populations (dark blue) achieve much worse results than the last populations (yellow), meaning that the optimization works well in these cases.

Boxplots in Figure 3.5 further emphasize the positive effect of the optimization. The data for boxplots were collected from six executions of the Crawler test case using the *MOGA-conf* configuration and the *fitness1a* function. The boxplots show a comparison of results achieved by the individuals from the initial population, which are generated randomly (de-




(a) WConcurPair (b) HBPair




(c) GoldiLockSC*

Figure 3.5: Comparison of Random vs. the last generation from MOGA in the Crawler test case using three considered objectives.

noted as *Random* in boxplots), and therefore represent values that can be achieved without the MOGA approach, and the results achieved by the individuals from the final population (denoted as *MOGA*). The bloxplots show data for the considered objective functions, i.e. *WConcurPair*, *HBPair*, and *GoldiLockSC**. The difference between the random and the MOGA approach is evident in all boxplots.

Finally, Table 3.5 summarizes our study of correlation and heritability for all three considered test cases (i.e. Airlines, Animator, and Crawler) and two fitness functions, namely, *fitness1a* and *fitness1b*. Note that a high correlation coefficient represents high heritability, which is good for MOGA. In general, the table shows very high correlation coefficients (i.e. above 0.8), but there are several important exceptions.

In some cases, for instance in the Animator test case and the *WConcurPair* coverage, the correlation coefficient for *fitness1a* is much higher (over 0.9) than for *fitness1b* (below 0.5). This is impact of difference between computation of cumulative value (*fitness1a*) and median (*fitness1b*). *WConcurPair* is a very detailed coverage metrics and therefore the non-determinism in its values is the highest from the considered metrics. In such cases, the cumulative value represents a somewhat stable value that provides better results in the end.

The situation with the *HBPair* coverage in the same test case is the opposite. In average, better results were achieved when median was used. This is because *HBPair* represents a coverage metrics with a high level of abstraction and hence a low level of diversity. The high level of abstraction is emphasized in the Airlines test case, in which the *NA* value is presented: the Airlines test case contains only a little synchronization; therefore, it was easy to achieve full coverage in this case. There was no improvement in this case.

Overall, the presented results show clear evidence that despite presence of a certain level of non-determinism in the evaluation of the individuals, the MOGA approach is able

Table 3.5: Parents-offspring correlation coeficients.

| | Coverage | Correlation coefficient fitness1a | fitness1b |
|---|---|---|---|
| Airlines | WConcurPair | 0.726 | 0.703 |
| | HBPair | NA | NA |
| | GoldiLockSC* | 0.812 | 0.458 |
| Animator | WConcurPair | 0.914 | 0.578 |
| | HBPair | 0.705 | 0.598 |
| | GoldiLockSC* | 0.609 | 0.564 |
| Crawler | WConcurPair | 0.873 | 0.906 |
| | HBPair | 0.964 | 0.960 |
| | GoldiLockSC* | 0.955 | 0.940 |

to search for better solutions. We therefore do not need to use other tools for reduction of non-determinism. This is a very positive finding, because most of the non-determinism reduction techniques require increase in the number of measurements (i.e. evaluations) [50]. Such solutions would lead to higher time requirements, as in our case, each evaluation is realized by an execution of SUT.

### 3.4.2 Fitness Functions Comparison

In this section, we focus on the ability of our modifications of the search process to avoid degradation of the search process implemented by our multi-objective genetic algorithms when used with the fitness functions proposed in Section 3.3 (i.e. *fitness1a*, *fitness1b*, and *fitness2*). Here, *degradation* refers to a situation when the population contains more than one copy of the same individual which implies a loss of diversity in the population.

The same test cases as above (namely Airlines, Animator, and Crawler) were employed for the purposes of this comparison. We used the same setting of MOGA as in the previous experiment: the *MOGAconf* configuration. Moreover, we randomly selected six initial populations and let MOGA start from these populations only. This allows us to compare the considered fitness functions on the same initial data.

The graph in Figure 3.6 summarizes the results obtained in the comparison. The graph shows how the average number of distinct individuals (y-axis) develops across generations (x-axis). The average values are computed from all executions of all three considered test cases.

The worst results from the newly proposed fitness functions were achieved by the *fitness1a* function, which considers cumulated values of objectives. The sparsity computation used by the *NSGA-II* algorithm described in Section 3.2 should avoid degeneration of the search process. Therefore, we were curious why the degeneration is happening here. The problem is caused by the non-deterministic evaluation of individuals. Further analysis shows that the same individual is evaluated quite differently and because sparsity is computed from the achieved results (i.e. objective vector), *NSGA-II* considers such an individual to be different form the already known ones. Moreover, such individuals are quite successful and therefore preferred by the algorithm. Therefore, the algorithm keeps

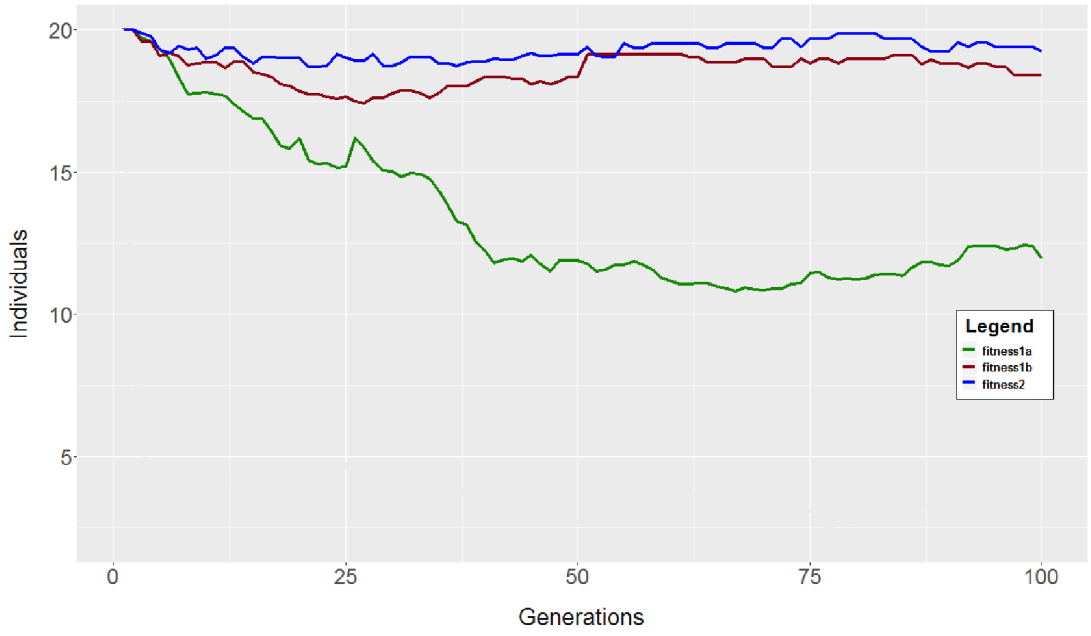



Figure 3.6: Degeneration of search process when using MOGA with proposed fitness functions.

the newly evaluated individual in the archive, which contains such an individual already—with a different evaluation.

Much better results were achieved by the *fitness1b* fitness function: this shows that the use of median indeed suppresses the non-deterministic evaluation (as described in Section 3.3). The graph clearly shows that this positively affects the quality of the search, because in average, *fitness1b* suffers from degeneration much less. The best results from the degeneration point of view were achieved by *fitness2*, which combines the success of suppressing non-determinism and objectives introducing less non-determinism (*Error*, *LockSet*).

## 3.5 Comparison with Single-Objectives Genetic Algorithm

An objective of this section is to show that our approach provides better results when compared to the sooner proposed use of the single-objective genetic approach (SOGA), which we already mentioned in Section 3.1.

The section presents results of four experiments comparing the proposed MOGA-based approach with the SOGA-based approach and both approaches with the random approach. First, we show difference between degeneration of the search process identified in the SOGA-based approach in [42] and in our MOGA-based approach which does not suffer from degeneration. Then, we show that the proposed penalization does indeed lead to a higher coverage of uncommon behaviour. Finally, we focus on a comparison of the MOGA, SOGA, and random approaches with respect to their efficiency and stability.

The experiments presented below were conducted on a set of eight concurrent benchmarks — Airlines, Animator, Crawler, Elevator, MolDyn, MonteCarlo, Raytracer, and Rover (see Section 2.6).

In the experiments, we used the settings of the MOGA that was already presented in Section 3.3.3 and Section 3.3.4, i.e., each candidate solution is evaluated 10 times, the achieved coverage is penalized, and the median values for the selected metrics are computed. Size of the population is 20, number of generations is 50, the crossover type is two, and the mutation probability is 0.5. As the SOGA-based approach uses *time* as one of the objectives in the fitness function, we added the execution time of tests variable to the MOGA fitness function for optimization of tests with small resource requirements. The objectives in the MOGA approach selected for following experiments are *GoldiLockSC**, *GoodLock**, *WConcurPairs*, and *Time*.

In the experiments, we use the following parameters of the SOGA-based approach taken from [42]: size of population 20, number of generations 50, two different selection operators (tournament among four individuals and fitness proportional[2]), the *any-point* crossover with probability 0.25, a low mutation probability (0.01), and two elites (that is 10 % of the population). However, to make the comparison more fair, we built the fitness function of the SOGA-based approach from the objectives selected above[3]:

$$\frac{WConcurPairs}{WConcurPairs_{max}} + \frac{GoodLock^*}{GoodLock^*_{max}} + \frac{GoldiLockSC^*}{GoldiLockSC^*_{max}} + \frac{time_{max} - time}{time_{max}}.$$

The maximal values of objectives were estimated as 1.5 times the maximal accumulated numbers we got in 10 executions of the particular test cases. As proposed in [42], the SOGA-based approach uses cumulation of results obtained from multiple test runs without any penalization of frequent behaviours.

All results presented in this section were tested by the statistical t-test with the significance level $\alpha = 0.05$, which specifies whether the achieved results for Random, MOGA, and SOGA are significantly different. In a vast majority of cases, the test confirmed a statistically significant difference among the approaches.

**Degeneration of the Search Process.** Degeneration, i.e. a lack of variability in population, is a common problem of population-based search algorithms. Figure 3.7 shows average variability of the MOGA-based and the SOGA-based approaches computed from the search processes on eight considered test cases. The x-axis represents generations. The y-axis shows numbers of distinct individuals in the generations (max. 20). The higher value the search process achieves, the higher variability; therefore, low degeneration was achieved. The Figure 3.7 clearly shows that our MOGA-based approach does not suffer from the degeneration problem unlike the SOGA-based approach.

Degeneration of the SOGA-based approach and, subsequently, its tendency to get caught in a local maximum (often optimizing strongly towards a highly positive value of a single objective, e.g. minimum test time, but almost no coverage) can in theory be resolved by increasing the amount of randomness in the approach. However, then it basically shifts towards random testing. An interesting observation (probably leading to the good results presented in [42]) is that even a degenerated population can provide a high coverage if

[2]Experiments presented in [42] showed that using these two selection operators is beneficial. Therefore, we used them again. On the other hand, for MOGA, the mating schema provides better results.

[3]In the experiments performed in [42], the fitness function was sensitive on weight. Therefore, we removed the weight from our new fitness function for SOGA.

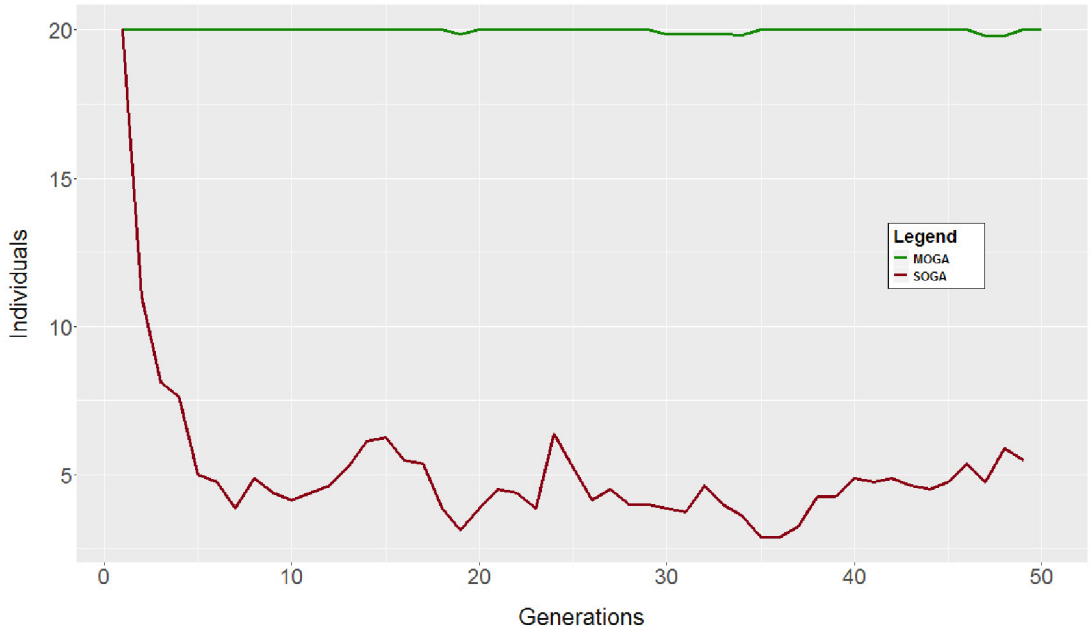



Figure 3.7: Degeneration of MOGA-based and SOGA-based search processes.

the repeatedly generated candidate solutions suffer from low stability, which allows them to test different behaviours in different executions.

Table 3.6: Impact of penalization built into MOGA approach.

| Test | MOGA | SOGA | Random |
|---|---|---|---|
| Airlines | 59.66 | **60.61** | 19.14 |
| Animator | 70.1 | **74.31** | 44.73 |
| Crawler | **70.73** | 66.32 | 61.19 |
| Elevator | **89.26** | 83.96 | 65.69 |
| Moldyn | **68.32** | 44.25 | 39.73 |
| Montecarlo | 40.13 | **54.52** | 28.25 |
| Raytracer | **73.08** | 60.49 | 54.68 |
| Rover | **53.87** | 41.45 | 30.62 |
| Average | **65.52** | 60.73 | 43.00 |

**Effect of Penalization.** The goal of the penalization scheme proposed above is to increase the number of tested uncommon behaviours. An illustration of the fact that this goal has indeed been achieved is provided in Table 3.6. The table particularly compares the results collected from 10 runs of the final generations of 20 individuals obtained through the MOGA-based and the SOGA-based approaches with the results obtained from 200 randomly generated individuals. Each value in the table gives the average percentage of uncommon behaviours spot by less than 50 % of candidate solutions, i.e. by less than 10 individuals. Number 60 therefore means that, on average, the collected coverage consists

of 40 % of behaviours that occur often (i.e. in more than 50 % of the runs) while 60 % are rare.

In most cases, if some approach achieved the highest percentage of uncommon behaviours under one of the coverage metrics, it achieved the highest numbers under the other metrics as well. Table 3.6 shows that our MOGA-based approach is able to provide a higher coverage of uncommon behaviours (where errors are more likely to be hidden) than the other considered approaches.

Table 3.7: Efficiency of considered approaches.

| Case | Metrics | MOGA | SOGA | Random |
|---|---|---|---|---|
| Airlines | C/Time | **0.06** | **0.06** | 0.04 |
| | S/Time | **3.73** | 3.29 | 2.98 |
| Animator | C/Time | 0.07 | **0.29** | 0.19 |
| | S/Time | 0.33 | **1.01** | 0.65 |
| Crawler | C/Time | 0.21 | **0.22** | 0.12 |
| | S/Time | **4.15** | 3.84 | 2.05 |
| Elevator | C/Time | 0.03 | **0.04** | 0.02 |
| | S/Time | 2.69 | **3.64** | 1.28 |
| Moldyn | C/Time | **0.01** | **0.01** | **0.01** |
| | S/Time | 11.73 | **16.83** | 2.56 |
| Montecarlo | C/Time | **0.01** | **0.01** | **0.01** |
| | S/Time | 9.52 | **9.66** | 0.01 |
| Raytracer | C/Time | **0.01** | **0.01** | **0.01** |
| | S/Time | **7.16** | 5.13 | 0.69 |
| Rover | C/Time | **0.11** | 0.10 | 0.08 |
| | S/Time | **5.17** | 2.49 | 2.18 |
| Avg. impr. | | 2.01 | 2.11 | |

**Efficiency of the Testing.** Next, we focus on the efficiency of the generated test settings, i.e. on their ability to provide a high coverage in a short time. We again consider 10 testing runs of the 20 individuals from the last generations of the MOGA-based and the SOGA-based approaches and 200 test runs under random generated test and noise settings. Table 3.7 compares the efficiency of these tests. To express the efficiency, we use two metrics: namely, *C/Time* shows how many coverage tasks of the *GoldiLockcSC** and *GoodLock** metrics got covered on average per a time unit (milisecond). *S/Time* indicates how many coverage tasks of the general purpose *WConcurPairs* coverage metric got covered on average per a time unit. Higher values in the table therefore represent higher average efficiency of the testing runs under the test settings obtained in one of the considered ways. The last row provides the average improvement (*Avg. impr.*) of the genetic approaches against random testing. Both genetic approaches are significantly better than the random approach. In some cases, the MOGA-based approach had a better evaluation, while the results were better for SOGA in some other cases. However, note that the MOGA-based approach is more likely to cover rare tasks (as explained in the previous paragraph). So even if it covers a comparable number of tasks with the SOGA-based approach, it is still likely to have more advantages from the practical point of view.

**Stability of Testing.** Finally, we show that candidate solutions found by the MOGA-based approach provide more stable results than the SOGA-based and the random approaches. For the MOGA-based and the SOGA-based approaches, Table 3.8 provides the average values of variation coefficients of the coverage under each of the three considered coverage criteria for each of the 20 candidate solutions from the last obtained generations across 10 test runs. For the case of random testing, the variation coefficients were calculated from 200 runs generated randomly. The last row of the table shows the average variation coefficient across all the case studies. The table clearly shows that our MOGA-based approach provides more stable results when compared to the other approaches.

Table 3.8: Stability of testing.

| Case | MOGA | SOGA | Random |
|---|---|---|---|
| Airlines | **0.06** | 0.17 | 0.29 |
| Animator | **0.02** | 0.11 | 0.12 |
| Crawler | 0.38 | 0.38 | **0.26** |
| Elevator | 0.50 | **0.48** | 0.58 |
| Moldyn | **0.11** | 0.20 | 0.70 |
| Montecarlo | 0.13 | **0.11** | 0.89 |
| Raytracer | **0.16** | 0.46 | 0.76 |
| Rover | **0.08** | 0.10 | 0.32 |
| Average | **0.18** | 0.25 | 0.49 |

### 3.5.1 Threats to Validity

Any attempt to compare different approaches faces a number of challenges, because it is important to ensure that the comparison is as fair as possible. The first issue to address is that of *internal validity*, i.e. whether there has been a bias in the experimental design or stochastic behaviour of the meta-heuristic search algorithms that could affect the obtained results. To attend to this issue, Section 3.3.2 provides a brief discussion and experimental evidence that supports the choice of the NSGA-II MOGA algorithm out of the three considered algorithms. To address the problem of setting various parameters of meta-heuristic algorithms, a number of experiments was conducted to choose configurations that would provide good results in the given context. Similarly, our choice of suitable objectives was done based on observations from the previous experimentation [61]. Care was taken to ensure that all approaches are evaluated in the same environment.

Another issue to address is that of *external validity*, i.e. whether there has been a bias caused by external entities, such as the selected case studies (that is, programs to be tested in our case) used in the empirical study. The diverse nature of programs makes it impossible to sample a sufficiently large set of programs. The chosen programs contain a variety of synchronization constructs and concurrency-related errors that are common in practice, but they represent a small set of real-life programs only. The studied execution traces conform to real unit and/or integration tests. As with many other empirical experiments in software engineering, further experiments are needed to confirm the results presented here.

# Chapter 4

# Using Data Mining in Testing of Concurrent Programs

As it has been already said, the problem with testing of concurrent programs is in the choosing suitable noise injection heuristics and suitable values of their parameters (as weel as suitable values of parameters of the programs being tested themselves). In this chapter, we propose the solution of this problem. Here, by suitable, we mean such settings that maximize chances of meeting a given testing goal (such as, e.g. maximizing coverage of rare behaviours and thus maximizing chances to find rarely occurring concurrency-related bugs). Our approach is, in particular, based on using data mining in the context of noise-based testing. We use the approach both to get more insight about the importance of the different heuristics in a particular testing context as well as to improve fully-automated noise-based testing (in combination with both random as well as genetically optimized noise setting).

## 4.1 Introduction

In this chapter, our approach is, in particular, based on using *data mining*, applied on a sample of test runs of a given concurrent program, to derive *classifiers* capable of distinguishing which test and noise settings are suitable and which unsuitable for the given testing goal. To be more precise, we use *decision trees* and the *AdaBoost* machine learning algorithm, which is a well-known technique for building high-quality classifiers.

We show how AdaBoost can be applied to gain new knowledge about efficient noise-based testing of a given concurrent program with a given testing goal (or even more generally for a class of programs and/or testing goals). Subsequently, we show how the results obtained by data mining can be used to fully automatically improve testing based on randomly set up noise injection. This is achieved by either filtering out unsuitable randomly chosen settings or by narrowing down the random generation to suitable ranges of noise and/or test case parameters. Moreover, we also show that the obtained results can be used to guide and consequently speed up an automated search-based process of finding suitable values of test and noise parameters. For that purpose, we combine the process of mining of suitable settings of noise-based testing with a subsequent genetic optimization restricted to the values considered as suitable by data mining.

In order to show that the proposed approach can indeed be useful, we apply it for optimizing the process of noise-based testing for two particular testing goals on a set of several benchmark programs. Namely, we consider the testing goals of *reproducing known*

*errors* and *covering rare interleavings* which are likely to hide so far unknown bugs. Our experimental results confirm that the proposed approach can discover useful knowledge about the influence and suitable values of test and noise parameters, which we show in two ways: (1) We manually analyze information hidden in the classifiers, compare it with our long-term experience from the field, and use knowledge found as important across multiple case studies to derive some new recommendations for noise-based testing. (2) We show that the obtained classifiers can be used—in a fully automated way—to significantly improve efficiency of noise-based testing using a random selection of test and noise parameters as well as to be successfully combined with finding suitable noise settings by genetic optimization.

## 4.2 Related Work

Below, we discuss works where data mining is applied in testing. None of them, however, is going in the same direction as the research presented in this thesis.

Most of the existing works on obtaining new knowledge from multiple test runs of concurrent programs focus on gathering debugging information that helps to find the root cause of a failure [25, 99]. In [99], a machine learning algorithm is used to infer points in the execution such that the error manifestation probability is increased when noise is injected into them. It is then shown that such places are often involved in the erroneous behaviour of the program. Another approach [25] uses a technique similar to data mining, more precisely, a feature selection algorithm, to infer a reduced call graph representation of the system under test, which is then used to discover anomalies in the behaviour of the system under test within erroneous executions.

None of the works above, and, to the best of our knowledge, no other existing work has applied data mining for finding values of test and noise parameters suitable for noise-based testing of concurrent programs. The only exception is our preliminary work [6], on which this chapter is based. However, compared with [6], the present chapter provides (1) a significantly improved presentation of the idea, (2) it proposes a new way of exploiting the results from data mining for fully-automated noise-based testing, (3) a combination of data mining with genetic approaches, and (4) it provides a significantly improved experimental evaluation of the approach.

Naturally, there is much richer literature and tool support for data mining test results without a particular emphasis on concurrent programs. The existing works study different aspects of testing, including identification of test suite weaknesses [1], optimisation of the test suite [106], or error localization [27]. Adler et al [1] show that a substring hole analysis is used to identify sets of untested behaviours using coverage data obtained from testing of large programs. Contrary to the analysis of what is missing in coverage data and what should be covered by improving the test suite, other works focus on what is redundant. Yoo et al [106] show that a clustering data mining technique is used to identify tests which exercise similar behaviours of the program. The obtained results are then used to prioritise the available tests. Erman et al [27] show that clustering of similar test case failures is used to help the analyst to identify the underlying causes of the failures and thus to make it easier to deal with huge numbers of test results obtained due to test automation.

Further, data mining techniques are, of course, used in many other areas of software engineering than testing. An exhaustive list of such applications is beyond the scope of this chapter, and so we mention just a few examples. For instance, in the recent result [105], machine learning is used to extract design knowledge allowing one to improve assignment of

responsibilities to classes, which is a vital task in object-oriented design. Cheung et al [14] show that clustering is used to detect smells in spreadsheet cells, which are susceptible to contain errors. Rubinič et al [86] show that machine learning is applied for software defect prediction, using ensembles of genetic classifiers to deal with imbalanced data sets. Luo et al [76], data mining is used for automatically identifying code changes that may potentially be responsible for a performance regression. Next, Kreutzer et al [56] show that a clustering algorithm is used in combination with two syntactical similarity metrics to automatically detect groups of similar code changes. Liang et al [70] focus on improving the precision of code mining with the aim of error detection by carefully preprocessing the source code. Tantithamthavorn et al [95] show that an automated parameter optimization technique has been applied to obtain prediction models in the form of classifiers trained to identify defect-prone software modules. Wang et al [101] show that machine learning is used to automatically learn a semantic representation of programs from their source code.

## 4.3 Preliminaries

In this section, we introduce the basics of the *AdaBoost* approach to machine learning for a proper understanding of the rest of the chapter. AdaBoost is at the heart of our approach to finding suitable values of noise parameters.

### 4.3.1 AdaBoost Machine Learning Algorithm

The core idea of our approach is to apply AdaBoost in noise-based testing to derive classifiers capable of distinguishing suitable and unsuitable settings of noise parameters as well as parameters of the programs under test (and consequently to facilitate searching for suitable test and noise settings). The AdaBoost algorithm, introduced in 1995 by Freund and Schapire [32, 33, 34], is a widespread machine learning technique based on improving ("boosting") the strength of multiple weak classifiers. This is achieved by weighting outputs of the weak classifiers and combining them into a single strong classifier. A weak classifier is any classifier that behaves better than random guessing (i.e. its error degree is less than 0.5 in the binary classification case).

AdaBoost works in iterations. In each iteration, the method aims at producing a new weak classifier in order to improve the precision of the so far constructed strong classifier. To construct the new classifier, objects in the training set are assigned weights. Initially, the weights are distributed uniformly. In each iteration, weights of wrongly classified objects are enlarged, which is then used in the next round to derive and add a new weak classifier focusing on the hard examples in the training set, hence improving the precision of the strong classifier.

In the binary classification case, the input of AdaBoost is a set $\mathcal{X} = \{(\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)\}$ where each $\overline{x}_i$ is an object from some space $\mathbb{X}$ of objects that we might want to classify as having or not having some property of interest, and each label $y_i$ belongs to the set $\mathbb{Y} = \{1, -1\}$, which says whether $\overline{x}_i$ does or does not have the property of interest. The input set $\mathcal{X}$ is then commonly split to two subsets—the training set $\mathcal{T}$ and the validation set $\mathcal{V}$. The training set is used to get a classifier while the validation set is used for evaluating the precision of the obtained classifier. More information about computing the precision is already presented in Section 2.4.1.

Moreover, in order to avoid over-fitting and to increase confidence in the obtained results, the process of choosing the training and validation set and of learning and validating

the classifier can be repeated several times, allowing one to judge the average values and standard deviation of accuracy and sensitivity. If the obtained classifier is not validated successfully, one can repeat the AdaBoost algorithm with more boosting iterations and/or a larger input set $\mathcal{X}$.

The final strong classifier is obtained in the form

$$F(\overline{x}) = sign\left(\sum_{i=1}^{T} w_i r_i(\overline{x})\right)$$

where $\overline{x} \in \mathcal{X}$, $T$ is the number of boosting iterations, $r_i$ is the weak classifier produced at the $i$-th iteration of the algorithm (producing decisions from the set $\mathbb{Y}$), and $w_i$ is a non-negative weight expressing confidence in the $i$-th weak classifier.

## 4.4 Classification-based Data Mining in Noise-based Testing

In this section, we describe our proposal of using a particular kind of AdaBoost classifiers for discovering which test and noise parameters and which of their values are the most influential for a given program under test and a given testing goal (or, even in general, across different programs under test and/or testing goals). We first describe the concrete kind of AdaBoost classifiers that we propose to be used in noise-based testing, and we provide a generic approach for deriving such classifiers. We then concretise the method for two concrete testing goals common in practice—namely, for finding rare behaviours in which so far unknown bugs may reside and for reproducing known errors. Subsequently, we discuss how the derived AdaBoost classifiers can be used to draw some conclusions about which test and noise configurations are the most influential in the given setting. Finally, we discuss three ways of using the derived classifiers in fully-automated testing.

### 4.4.1 Combining Data Mining Based on AdaBoost with Noise-based Testing

For our application of data mining with the aim of finding suitable settings of noise-based testing of concurrent programs, we propose using data mining based on *binary classification*. Methods that have been used for binary classification in the literature include decision trees, Bayesian networks, support vector machines, or neural networks [103]. In this work, we, in particular, choose *decision trees*. This is motivated by the fact that one can easily understand and further exploit information hidden in decision trees obtained by machine learning, which we leverage in the following.

*Decision trees*, such as those shown in Fig. 4.1, can be viewed as hierarchically structured decision diagrams whose nodes are labelled by Boolean conditions on the items to be classified and whose leaves represent classification results (in our case, $+1$ is used to denote a positive result, while $-1$ denotes a negative result). The decision process starts in the root node by evaluating the condition associated with the root on the item to be classified. According to the evaluation of the condition, a corresponding branch is followed into a child node. This descent, driven by the evaluation of the conditions assigned to the encountered nodes, continues until reaching a leaf node, and hence a decision. Decision trees are usually employed as a predictive model constructed via a decision tree learning procedure, which uses a training set of classified items.

In order to reduce the natural tendency of decision trees to be unstable (meaning that a minor data oscillation can lead to a large difference in the classification), we combine

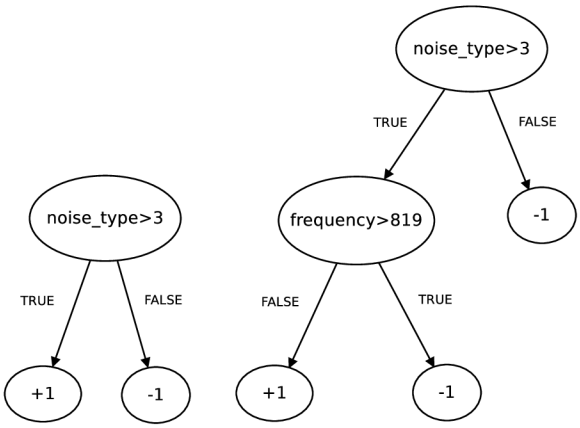



Figure 4.1: Examples of decision trees.

them with using the AdaBoost approach described in Section 4.3.1. Decision trees, with the classification result being 1 or $-1$, are used as the weak classifiers. The resulting strong classifier then consists of a set of weighted decision trees that are all applied on the item to be classified, their classification results are weighted by the appropriate weights, summarized, and the sign of the result provides the final decision.

In order to be able to apply AdaBoost in noise-based testing, one has to first define some *testing goal* expressible as a binary *test property* that can be evaluated over test results such that both positive and negative answers are obtained. The test property will typically be based on some non-binary *test quantity* such as the number of discovered error occurrences, number of covered tasks of some metric, testing time, or a (weighted) combination of such quantities. The binary test property can then be obtained by taking the median value of the test quantities obtained throughout the test runs and by classifying test and noise settings to those that lead (or do not lead) to results above the median.

**Example 4.4.1.** *So, a binary test property can, e.g. look like coverage* $> C \wedge$ *time* $< T$ *where coverage measures coverage under the chosen coverage metric,* $C$ *is the median coverage obtained in the so far performed test runs, test measures the time of executing a test, and* $T$ *is the median testing time in the so far performed runs.*

The requirement of having both positive and negative results can be a problem in some cases, notably in the case of discovering rare errors where getting positive results is—naturally—very rare. In such a case, one has to use a property that approximates the target test property (e.g. by replacing the discovery of rare errors by discovering any rare program behaviours even when they do not contain an error) and provides both positive and negative answers sufficiently often. Of course, once some testing goal is satisfied (e.g. once testing aimed at rare behaviours manages to find some error), another testing goal can become more urgent—e.g. that of repeatedly reproducing the same error for debugging purposes or finding other similar errors. The training process is then to be repeated, possibly using the newly available test results found by previously conducted test runs.

Further, note that, in the context of testing concurrent programs, the test property will typically not be defined over results of particular test runs but rather on results of multiple test runs performed under the same test and noise setting. The reason is the need of minimizing the influence of *scheduling non-determinism*. The results obtained in several test

runs can be summarised by taking, e.g. the median or cumulative value of the considered test quantity.

Once the test property representing the chosen testing goal is defined, a number of test and noise configurations is to be generated at random. Several test runs are to be performed for each of these configurations, and the test property is to be evaluated on each of the series of the test runs performed with the same test and noise configuration. For each of the considered test and noise configurations, a couple $(\overline{x}, y)$ is formed where $\overline{x}$ is a vector recording the test and noise configuration used and $y \in \{1, -1\}$ is the result of evaluating the test property. This way, we obtain the set $\mathcal{X} = \{(\overline{x}_1, y_1), \ldots, (\overline{x}_n, y_n)\}$ to be used as the input of AdaBoost as described in Section 4.3.1.

**Example 4.4.2.** *An example of a couple, which can appear in the set $\mathcal{X}$ if we consider three noise parameters, e.g. noise frequency, strength of noise, and type of noise, can be $((839, 28, 1), -1)$. It says that for the values 839, 28, and 1 of the noise frequency, strength of noise, and type of noise, respectively, the test property evaluated negatively.*

In Section 4.4.2, we illustrate and further concretise the above ideas by proposing concrete test properties and ways of evaluating them for two testing goals common in practice: namely, *finding rare behaviours* and repeatedly *reproducing known errors*.

Once the set $\mathcal{X}$ is obtained, the AdaBoost algorithm can be applied and the result validated as described in Section 4.3.1. A successfully validated classifier can subsequently be analyzed to get some insight which test and noise parameters are influential for testing the given program and which of their values are promising for meeting the defined testing goal. Such knowledge can then in turn be used by testers when thinking of how to optimize the testing process. We propose a way how such an analysis can be done in Section 4.4.3, and we experiment with it in Section 4.5.4. Moreover, the obtained classifier can also be used to fully automatically improve performance of noise-based testing: we propose three approaches how this can be done (two of these approaches based on filtering randomly generated test and noise settings and one based on a combination with genetic optimization) in Section 4.4.4. Experiments with these approaches are then described in Section 4.5.5.

### 4.4.2 Finding Rare Behaviours and Reproducing Known Errors

We now concentrate on two concrete testing goals: namely, (1) *repeatedly finding known errors*, which is useful for debugging purposes, and (2) *finding rare behaviours*, which is useful for finding bugs missed by common testing runs. For these two different goals, we propose concrete test properties and a way of evaluating them that turned out as suitable in our experiments for deriving input sets for AdaBoost such that AdaBoost in turn produces appropriately trained classifiers for the given testing goals.

In the case of trying to repeatedly reproduce a known error, the test property of interest is simply the *error manifestation property* that indicates whether an error manifested during the performed test executions or not. When deriving the input set $\mathcal{X}$ for AdaBoost that should in turn produce a classifier suitable for reproducing the given error, we generate a number of random test and noise configurations, perform several test runs with each of the configurations[1], and compute the number of test runs in which the error has been found. Then, we compute the median value of the number of runs in which an occurrence of the given error has been found for the different considered test and noise configurations. Configurations that reached a number of error occurrences above the median are marked

[1] In our experiments, we, in particular, use five runs.

as positive whereas the remaining ones are marked as negative. This will give us the set $\mathcal{X}$ that will be split into a testing set and a validation set. The testing set will be used as the input for AdaBoost, which will then produce an appropriately trained classifier for the error manifestation property.

**Example 4.4.3.** *For an example of getting an input set for AdaBoost according to the above description, see Table 4.1. In particular, we consider five combinations of values of three noise parameters, namely, noise frequency, strength of noise, and type of noise. Assume that when we perform five testing runs with each of the settings, we get the number of error manifestations shown in the fourth column of the table—with the median number of error manifestations being* 0*. Then the classification results will be those given by the fifth column of the table. This gives us the set* $\mathcal{X} = \{((839, 28, 1), 1), ((114, 36, 5), -1), ((724, 48, 4), -1), ((895, 12, 0), 1), ((234, 8, 4), -1)\}$ *that will be split into a training and validation set for AdaBoost.*

Table 4.1: An example of constructing an input for AdaBoost for the error manifestation property.

| *noise frequency* | *strength of noise* | *type of noise* | *number of error manifestations* | *classification result* |
|---|---|---|---|---|
| 839 | 28 | 1 | 2 | 1 |
| 114 | 36 | 5 | 0 | -1 |
| 724 | 48 | 4 | 0 | -1 |
| 895 | 12 | 0 | 5 | 1 |
| 234 | 8 | 4 | 0 | -1 |

Once a classifier is derived, its precision and stability are tested on the validation set. In particular, we let the generated configurations be classified by the derived classifier as suitable or unsuitable for reproduction of the known errors, and, subsequently, we check correctness of the classification through repeated test runs under these configurations. The concrete numbers of test runs considered to get the training and validation sets in our experiments are provided in Sections 4.5.3 and 4.5.5.

Next, we consider the case of finding test and noise configurations suitable for testing rare behaviours in which so far unknown bugs might reside. In order to achieve this goal, we use classification according to a *rare events property* that indicates whether a test execution covers at least one rare coverage task of a suitable coverage metric—in our experiments, the *GoldiLockSC*$^*$ metric [26] is used for this purpose. To distinguish rare coverage tasks, we collect the tasks that were covered in at least one of the performed test runs (i.e. both from the training and validation sets), and, for each such coverage task, we count the frequency of its occurrence in all of the considered runs. We define the rare tasks as those that occurred in less than 20 % of the test executions.

Furthermore, when learning the classifier, we want to avoid the scenario where we find some test and noise configurations that are capable of finding some behaviours that are rare in normal test runs, but they lead to discovering the same behaviours in each noised test run again and again. This is, we ideally want to keep finding different rare behaviours in each test run. To stress this goal, we focus on the *cumulative number* of covered rare

tasks, not only on coverage in individual executions. In our experiments, we, in particular, use cumulation from five test runs. This is, we randomly generate a number of test and noise configurations. With each of them, we execute five test runs, and we cumulate (i.e. unite) the sets of covered rare tasks.

Subsequently, as we consider the time needed for testing to be also important, we take the sizes of the cumulated sets of covered rare tasks and divide them by the time needed to perform the considered five test executions. We take as positive the test and noise configurations whose cumulated number of covered rare tasks divided by the needed test time is above the median value of this combined test quantity. We then derive the AdaBoost classifier and test its precision and stability. The concrete numbers of test runs considered to get the training and validation sets in our experiments are again provided in Sections 4.5.3 and 4.5.5.

**Example 4.4.4.** *Table 4.2 gives an example of obtaining an input set for AdaBoost for the rare behaviours property according to the above description. Namely, we consider three combinations of three noise parameters (noise frequency, strength of noise, and type of noise as before). To shorten the example, we assume that three testing runs were performed with each of these configurations only. Further, we assume that the rare tasks that were covered in the testing runs are as shown in the fourth column. The fifth column gives the time we assume to be consumed for the testing runs. The sixth column then gives the corresponding cumulative coverage of rare tasks divided by the total consumed time. Finally, the last column gives the appropriate classification result (due to the median coverage being* $3/7$*). The value from the last column is to be used together with the values in the first three columns to derive the input set for AdaBoost:* $\mathcal{X} = \{((83, 28, 1), -1), ((451, 44, 3), 1), ((729, 32, 3), -1)\}$.

Table 4.2: An example of constructing an input for AdaBoost for the rare behaviours property.

| *noise freq.* | *noise strength* | *noise type* | *covered rare tasks* | *testing time* | *cumulative coverage per time* | *classification result* |
|---|---|---|---|---|---|---|
| 83 | 28 | 1 | run 1: $\{a, c, d\}$ | 3 | $(\lvert\{a, c, d\}\rvert = 3)/7$ | -1 |
| | | | run 2: $\{a, d\}$ | 2 | | |
| | | | run 3: $\{c, d\}$ | 2 | | |
| 451 | 44 | 3 | run 1: $\{a, d\}$ | 2 | $(\lvert\{a, c, d, e\}\rvert = 4)/5$ | 1 |
| | | | run 2: $\{c, e\}$ | 2 | | |
| | | | run 3: $\{d, e\}$ | 1 | | |
| 729 | 32 | 3 | run 1: $\{c, e\}$ | 3 | $(\lvert\{c, e\}\rvert = 2)/8$ | -1 |
| | | | run 2: $\{c\}$ | 2 | | |
| | | | run 3: $\{e\}$ | 3 | | |

### 4.4.3 Analysing Information Hidden in Classifiers

In order to be able to easily analyze information hidden in the classifiers generated by AdaBoost, we have decided to restrict the height of the basic decision trees used as weak classifiers to one. Moreover, our experiments showed us that increasing the height of the weak classifiers does not lead to significantly better classification results.

A decision tree of height one consists of a root labelled by a condition concerning the value of a single test or noise parameter and two leaves that correspond to the cases when the condition is or is not satisfied and that are labelled as leading to either positive or negative classification. AdaBoost provides us with a set of such trees, each with an assigned weight. For better understanding which parameters are important for testing, we convert this set of trees into a set of rules such that we get a single rule for each test or noise parameter that appears in at least one decision tree. The rules consist of a condition and a weight. In particular, the conditions have the form of a conjunction of interval constraints, and the weights are real numbers from the range between zero and one.

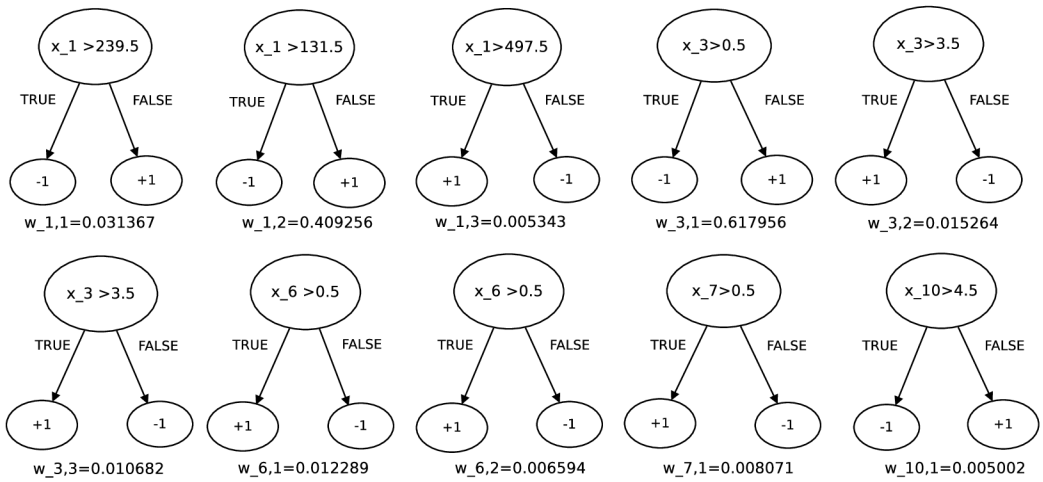



Figure 4.2: An example of several decision trees with conditions over parameters $x_1, x_3, x_6, x_7$ and $x_{10}$ created by the AdaBoost algorithm.

The rules are obtained as follows. First, decision trees with negative or zero weights are omitted because they correspond to weak classifiers with the weighted error greater or equals to 0.5. Next, the remaining decision trees are grouped according to the parameter about whose value they speak. To illustrate the above, assume that AdaBoost gives us, e.g. the ten decision trees with positive weights that are shown in Fig. 4.2. For each obtained group of the trees, a single rule is produced by taking the disjunction of the interval constraints associated with the grouped decision trees[2]. Intuitively, taking the disjunction corresponds to the fact that each of the intervals was found to be relevant for the given testing goal. The weight of the rule is computed by summarizing the weights of the trees from the concerned group and normalising the result by dividing it by the sum of the weights of all trees from all groups. This is, if all decision trees with positive weights created by AdaBoost are $w_1, \ldots, w_m$, and the concerned group $G$ consists of $n \leq m$ trees with weights $w_{i_1}, \ldots, w_{i_n}$ where $\forall 1 \leq j \leq n : \ 1 \leq i_j \leq m$, then the weight of the rule created from $G$ will be computed as the fraction $\frac{\sum_{j=1}^{n} w_{i_j}}{\sum_{k=1}^{m} w_k}$.

In our example, we focus on the importance of the different parameters. We start with parameter $x_1$. For this parameter, when we take the disjunction of the interval constraints associated with the trees corresponding to $x_1$ (i.e. the first three trees in Fig. 4.2), we obtain the condition $x_1 \leq 239.5 \vee x_1 \leq 131.5 \vee x_1 > 497.5$, which can be simplified to $x_1 \leq 239.5 \vee x_1 > 497.5$. The weight of this rule is given by the sum of the three concerned trees

[2]In particular, the interval constraint of the tree is taken as is when the true branch of the decision tree leads to the +1 leaf. Otherwise, its complement must be taken.

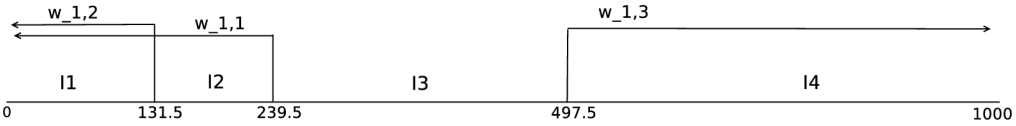



Figure 4.3: Division of values of parameter $x_1$ to intervals associated with the decision trees from Fig. 4.2.

divided by the sum of the weights of all the trees in the figure, which gives the (rounded) weight $w_{x_1} = 0.398$. If we process the other parameters in the same way, we get the following weights: $w_{x_3} = 0.574$, $w_{x_6} = 0.017$, $w_{x_7} = 0.007$, $w_{x_{10}} = 0.004$. Note that the weights of the parameters satisfy the constraint $\sum_i w_{x_i} = 1$. Clearly, parameters $x_3$ and $x_1$ appear to have the highest importance in the given setting; parameters $x_6$, $x_7$, and $x_{10}$ appear to have at least some significance; while parameters such as $x_2$ are of no importance (since they did not even appear in any of the decision trees with positive weights).

From the rules obtained as described above, we can easily identify the parameters that most affect testing of the given program with the given testing goal. For that, we can simply take the parameters that are associated with the rules with the highest weights. In case we want to derive more general results—spanning over multiple testing goals and/or multiple tested programs, we can do that by looking for parameters (or values) that appear among the most influential ones among all (or most) of the considered test cases. Alternatively, one can also unite the training sets obtained for the different testing goals and/or programs under test, and then apply AdaBoost to the combined training set. In our example, the parameter which most affects the testing process is the parameter $x_3$ that has the highest weight.

Moreover, we can also see which concrete values of the different parameters are the most influential. In particular, assume that the condition of the rule derived for some parameter was created from a set $\mathcal{I} = \{I_1, ..., I_n\}$ of interval constraints where the decision trees that were associated with these intervals had weights $w_1, ..., w_n$. We identify all maximum subsets $\mathcal{J} = \{I_{i_1}, ..., I_{i_m}\} \subseteq \mathcal{I}$ of intervals with non-empty intersections (i.e. such that $\cap_{j \in \{1,...,m\}} I_{i_j} \neq \emptyset$) and assign each such set a weight $w_{\mathcal{J}}$ given by the sum of the weights of its elements, i.e. $w_{\mathcal{J}} = \sum_{j \in \{1,...,m\}} w_{i_j}$. Intuitively, the weights of all the decision trees whose interval constraints overlap contribute to the weight of their overlapping part. The most influential values of the given parameter are then given by the sets $\mathcal{J}$ with the highest weights—namely, by the union $\cup_{\mathcal{J}} \cap_{I \in \mathcal{J}} I$ of the intersections of the intervals $I$ belonging to the subsets $\mathcal{J}$ with the highest weights $w_{\mathcal{J}}$.

Thus, in the example, we have a look at the most influential values of some of the parameters from Fig. 4.2. In particular, we concentrate on parameter $x_1$. The parameter is associated with three decision trees and hence three interval constraints, which are illustrated in Fig. 4.3. From the illustration, we see that there are two maximum subsets of the interval constraints with non-empty intersections, namely, $\mathcal{J}_1 = \{x_1 \leq 239.5, x_1 \leq 131.5\}$ and $\mathcal{J}_2 = \{x_1 > 497.5\}$. The corresponding intersections are $x_1 \leq 131.5$ and $x_1 > 497.5$ with the (rounded) weights $w_{\mathcal{J}_1} = 0.441$ and $w_{\mathcal{J}_2} = 0.005$. Clearly, values of $x_1$ less than or equal to 131.5 are the most influential. In case one would like to have a finer look at the influence of the different values, one can take all subsets of the set of intervals associated with the given parameter, compute the corresponding intersections of the constraints and their weights (as in the case of the maximum subsets), and obtain a histogram of the weights—such as the one shown in Fig. 4.4 for the parameter $x_1$.

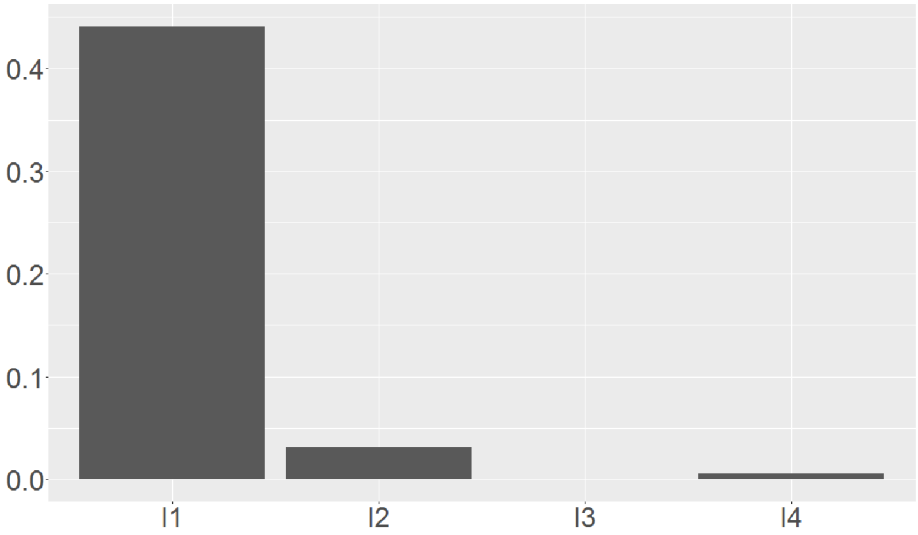



Figure 4.4: Histogram of weights of values of parameter $x_1$ derived from the decision trees in Fig. 4.2.

### 4.4.4 Using AdaBoost in Fully-Automated Testing

We now present several approaches of using AdaBoost for fully-automated noise-based testing. First, we describe two ways of combining AdaBoost with random generation of test and noise parameters. Second, we show how it can be combined with genetic algorithms for finding the most suitable values of test and noise parameters.

#### AdaBoost-Improved Random Testing

In practice, noise-based testing is often used with randomly generated test and noise configurations. The simplest way of using AdaBoost to improve on this practice is the following. When performing repeated test runs of a given program to meet a given testing goal, one can run the program with randomly generated test and noise configurations, but use only those randomly generated configurations that get classified as suitable by an AdaBoost classifier derived for the given program and testing goal as described in Subsection 4.4.2. This idea, considered already in our preliminary work [6], is rather simple, but it can provide quite nice results as we illustrate through our experiments presented in Subsection 4.5.5.

While the above approach can provide useful results, we now propose yet another way of combining AdaBoost with random generation of test and noise configurations, which was not considered in [6]. This approach is motivated by our observation that, in many of the case studies that we conducted and which we report later on, some test and noise parameters were significantly more important than others, even though the latter parameters were still influential. In such cases, however, the above proposed use of AdaBoost can include among useful test and noise configurations even some of those configurations where the less important parameters are set in a rather unsuitable way, which is tolerated due to the much higher weight of the more important parameters.

To improve on the above situation, we propose to build on the method for determining the most suitable values of each parameter, which is described at the end of Section 4.4.3. We then derive the test and noise configurations to be used by independently choosing

the value of each of the parameters at random but from the most suitable range of its values only. For instance, assume that we have test and noise parameters $x_1$, $x_2$, and $x_3$, and the approach of Section 4.4.3 tells us that their most influential values are from intervals $I_1$, $I_2$, and $I_3$, respectively. Then, every time we need a test and noise configuration for a repeated test run, we generate it as a three-tuple whose first item is randomly chosen from the interval $I_1$, the second item is randomly chosen from $I_2$, and the third one is randomly chosen from $I_3$. Our experiments presented in Section 4.5.5 show that this approach can indeed provide significantly better results than the first mentioned approach.

#### Combination of Genetic Algorithms and AdaBoost

Finally, we also propose a combination of using AdaBoost and the genetic algorithms that we considered for finding suitable test and noise configurations in chapter 3 (MOGA and SOGA approaches). This approach is motivated as follows. Chapter 3 showed that genetic algorithms can achieve very good results in finding suitable test and noise configurations, especially when trying to increase the achieved concurrency coverage, but they need to execute a huge number of test runs to get these configurations. The reason of this is that the genetic algorithms start with random initial configurations in the first generations and slowly create configurations with better results in the next generations. Our idea is to accelerate this process by restricting the range of possible values of the different test and noise parameters in which the genetic algorithms will search. In particular, we restrict the range of the parameters to the most influential values found through AdaBoost and the approach described at the end of Section 4.4.3. Thus, essentially, we use AdaBoost to get coarse knowledge on the suitable values of the test and noise parameters, and then we refine this knowledge using genetic algorithms. Our experiments presented below confirm that this approach can often significantly outperform all the other mentioned approaches.

## 4.5 Experimental Evaluation

In this section, we describe the experiments that we conducted to evaluate the approaches proposed above. We first provide a brief description of the benchmark programs that we used in our experiments. Next, we briefly characterize the accuracy and sensitivity of the AdaBoost classifiers that we were able to obtain for our case studies and testing goals. Subsequently, we analyze the knowledge hidden in the classifiers that we obtained, compare it with our experience obtained in other ways, and derive several new insights about the importance of the different test and noise parameters. Finally, we proceed to experiments illustrating that AdaBoost combined with genetic algorithms can also be quite successfully used in fully-automated noise-based testing.

### 4.5.1 Case Studies

For our experimental evaluation, we used the multi-threaded programs presented in Section 2.6. The first five of them contain known concurrency-related errors, and so they are suitable for experiments with reproduction of known bugs for debugging purposes. The remaining programs do not contain any known errors, and so they are added to the first five case studies within our experiments targeted at increasing coverage of rare behaviours[3].

[3]The case studies we present in this chapter do not include large programs due to we need to perform a rather large number of experiments with different test and noise settings: Already with the use cases we

### 4.5.2 Considered Test and Noise Parameters

In Section 4.4.1, we said that our input set $\mathcal{X}$ for AdaBoost will consist of couples $(\overline{x}, y)$ where $\overline{x}$ is a vector recording the test and noise configuration used and $y \in \{1, -1\}$ is the result of evaluating the considered test property. In our experiments, we—in particular—consider vectors $\overline{x}$ of test and noise parameters consisting of 12 entries, i.e. $\overline{x} = (x_1, x_2, \ldots, x_{12})$.

In our vectors of test and noise parameters, the parameter $x_1 \in \{0, \ldots, 1000\}$ represents the *noise frequency*, the parameter $x_2 \in \{0, \ldots, 100\}$ is the *strength of noise*, the parameter $x_3 \in \{0, \ldots, 5\}$ selects one of the six available basic noise seeding heuristics. The parameters $x_4, x_5 \in \{0, 1\}$ disable or enable the additional noise seeding heuristics *haltOneThread* and *timeoutTamper*, respectively.

The parameter $x_6 \in \{0, 1, 2\}$ controls the way how the *sharedVarNoise* noise placement heuristic behaves—namely, whether it is disabled ($x_6 = 0$), it applies the *sharedVarNoise-one* strategy injecting the noise at accesses to one randomly selected shared variable ($x_6 = 1$), or it applies the *sharedVarNoise-all* strategy inserting the noise at accesses to all shared variables ($x_6 = 2$). The parameter $x_7 \in \{0, 1\}$ disables or enables the *nonVariableNoise* heuristic. The parameters $x_8, x_9 \in \{0, 1\}$ disable or enable the *coverage-based* noise placement heuristic and the related *coverage-based-frequency* heuristic, respectively.

Finally, we summarize the parameters used by the above test cases (on top of the parameters of the noise injection technology itself) and explain in more detail their encoding in our experiments. These parameters are encoded as the parameters $x_{10} \in \{1, \ldots, 10\}$ and $x_{11}, x_{12} \in \{1, \ldots, 100\}$ in the experiments. In particular, *Animator*, *Cache4j*, *HEDC*, and *Crawler* are not parametrized, and hence $x_{10}$, $x_{11}$, $x_{12}$ are not used with them. In the *Airlines*, *Elevator*, *Montecarlo*, and *Raytracer* test cases, the $x_{10}$ parameter controls the number of the threads used. In the Rover test case, the $x_{10} \in \{1, \ldots, 7\}$ parameter selects one of the available test scenarios. The *Sor* and *TSP* test cases have two test parameters. The $x_{10}$ parameter is the number of iterations for *Sor* while it selects one of the available test scenarios for *TSP*. The $x_{11}$ parameter controls the number of the threads used for both of these test cases. The *Airlines* test case uses the $x_{11}$ and $x_{12}$ parameters where the $x_{11}$ controls how many cycles the test does and the $x_{12}$ parameter indicates the flight capacity.

The total number of noise configurations that one can obtain from the above can be computed by multiplying 1001 values of *noise frequency*, by 101 possible values of *noise strength*, the number of the basic noise seeding heuristics, which is six, by two to reflect whether *haltOneThread* is or is not used, two to reflect whether *timeoutTamper* is used, two to reflect whether the *nonVariableNoise* heuristic is used, two to reflect whether the *coverage-based* noise placement is used, two to reflect whether the *covergage-based-frequency* heuristic is used, and three to reflect the possible use case scenarios of the *sharedVarNoise* heuristic. This gives a rough estimate of about 58.2 million combinations of noise settings when we simplify the situation by ignoring the fact that some of the settings do not make sense when used together (for instance, enabling *coverage-based-frequency* heuristic has no effect when *coverage-based* heuristic is disabled). Of course, the state space of the test and noise settings then further grows with the possible values of parameters of the test cases and the testing environment [44].

consider, the experiments presented below took approximately 5,592 core hours, i.e. 233 core days. However, works such as [22] show that noise-based testing can be successfully used even on programs with millions of lines of code and can find previously unknown errors in complex industrial code.

### 4.5.3 Accuracy and Sensitivity of Classifiers

We now present data about the accuracy and sensitivity of the AdaBoost classifiers that we derived for the above test cases. For the first five of them that contain known concurrency errors, we have considered both the testing goal of reproducing a known error as well as the goal of increasing coverage of rare behaviours. For the remaining test cases, we have considered the latter goal only.

In our experiments, we used the implementation of AdaBoost available in the GML AdaBoost Matlab Toolbox[4]. We set it to use decision trees of height restricted to one and to use 10 boosting phases. When deriving the classifiers, we proceeded as described in Section 4.4.2. When deriving classifiers for the error manifestation property, we used 2000 random test and noise configurations. For the rare events property, due to a higher time-consumption of the experiments, we used 200 random test and noise configurations. To obtain data allowing us to derive the accuracy and sensitivity of the derived classifiers, 100 different random divisions of the randomly generated configurations to training and validation sets were considered.

Table 4.3: The average and standard deviation of the accuracy and sensitivity of the AdaBoost classifiers derived for the test cases containing known errors.

| | *Error reproduction* | | | | *Rare behaviours* | | | |
|---|---|---|---|---|---|---|---|---|
| | Accurancy | | Sensitivity | | Accurancy | | Sensitivity | |
| *CaseStudies* | Mean | Std | Mean | Std | Mean | Std | Mean | Std |
| Airlines | 0.7488 | 0.0163 | 0.8917 | 0.0250 | 0.6601 | 0.0508 | 0.6880 | 0.0900 |
| Animator | 0.8353 | 0.0154 | 0.9489 | 0.0195 | 0.8503 | 0.0489 | 0.9006 | 0.0549 |
| Crawler | 0.9916 | 0.0026 | 0.9948 | 0.0018 | 0.7453 | 0.0437 | 0.7549 | 0.0740 |
| Elevator | 0.9568 | 0.0056 | 0.9965 | 0.0034 | 0.7161 | 0.0439 | 0.7327 | 0.0797 |
| Rover | 0.8859 | 0.0142 | 0.9611 | 0.0088 | 0.6108 | 0.0406 | 0.6330 | 0.0950 |
| Average | 0.8837 | 0.0108 | 0.9586 | 0.0117 | 0.7165 | 0.0456 | 0.7418 | 0.0787 |

Table 4.4: The average and standard deviation of the accuracy and sensitivity of the AdaBoost classifiers derived for the test cases without known errors.

| | *Rare behaviours* | | | |
|---|---|---|---|---|
| | Accurancy | | Sensitivity | |
| *CaseStudies* | Mean | Std | Mean | Std |
| Cache4j | 0.8454 | 0.0671 | 0.8963 | 0.0907 |
| HEDC | 0.7819 | 0.0443 | 0.7797 | 0.0758 |
| Montecarlo | 0.6692 | 0.0607 | 0.6702 | 0.1230 |
| Raytracer | 0.6298 | 0.0713 | 0.6380 | 0.1114 |
| Sor | 0.7807 | 0.0457 | 0.8203 | 0.0797 |
| TSP | 0.6420 | 0.0674 | 0.6587 | 0.1179 |
| Average | 0.7248 | 0.0594 | 0.7439 | 0.0998 |

Tables 4.3 and 4.4 summarise the average accuracy and sensitivity of the derived AdaBoost classifiers and their standard deviations. One can clearly see that both the average

[4] http://graphics.cs.msu.ru/en/science/research/machinelearning/AdaBoosttoolbox

accuracy and sensitivity are quite high for the error reproduction test goal—with the average values being 0.8837 and 0.9586, respectively. For the testing goal of finding rare behaviours, both of the statistics have smaller values. However, the experiments presented in Section 4.5.5 show that the method works nicely even in their case. Moreover, the standard deviation is very low in all cases, which indicates that we always obtained results that provide meaningful information about our test runs.

### 4.5.4 Analysis of Knowledge Hidden in Obtained Classifiers

We now employ the approach described in Section 4.4.3 to interpret the knowledge hidden in the classifiers that we inferred for our test cases. From these classifiers, using the approach of Section 4.4.3, we derived the rules shown in Tables 4.5 and 4.6 for the error manifestation property and the rare behaviours property, respectively. For each test case, the tables contain a row whose upper part contains the condition of the rule (in the form of an interval constraint), and the lower part contains the appropriate weight from the interval $(0, 1)$.

In order to interpret the obtained rules, we first focus on rules with the highest weights (corresponding to parameters with the biggest influence). Then we look at the parameters which are present in rules across the test cases (and hence seem to be important in general) and parameters that are specific for particular test cases only. Next, we pinpoint parameters that do not appear in any of the rules and therefore seem to be of a low relevance in general.

Table 4.5: Inferred rules for the error manifestation property with the most influential intervals marked out.

Airlines

| Rules | $x_1 \leq 275$ | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3$ | $x_6 \leq 1.5$ | $2.5 < x_{10}$ | $73.5 < x_{12}$ |
|---|---|---|---|---|---|
| Weights | 0.16 | 0.50 | 0.04 | 0.18 | 0.12 |

Animator

| Rules | $705 < x_1$ | $2.5 < x_3 \leq 3.5$ | $x_6 \leq 0.5$ |
|---|---|---|---|
| Weights | 0.19 | 0.55 | 0.26 |

Crawler

| Rules | $x_1 \leq 215$ | $15 < x_2$ | $1.5 < x_3 \leq 3.5$ or $\mathbf{4.5 < x_3}$ | $0.5 < x_4$ | $x_5 \leq 0.5$ | $x_6 \leq 1.5$ |
|---|---|---|---|---|---|---|
| Weights | 0.32 | 0.1 | 0.38 | 0.05 | 0.08 | 0.07 |

Elevator

| Rules | $x_1 \leq 5$ | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$ | $x_7 \leq 0.5$ | $8.5 < x_{10}$ |
|---|---|---|---|---|
| Weights | 0.93 | 0.04 | 0.01 | 0.02 |

Rover

| Rules | $515 < x_1$ | $2.5 < x_3 \leq 3.5$ | $0.5 < x_4$ | $x_6 \leq 0.5$ |
|---|---|---|---|---|
| Weights | 0.21 | 0.48 | 0.08 | 0.23 |

As for the error manifestation property (i.e. Table 4.5), the most influential parameters are $x_3$ in four of the test cases and $x_1$ in the *Crawler* test case. This indicates that the selection of a suitable noise type ($x_3$) or noise frequency ($x_1$) is the most important decision to be done when testing these programs with the aim of reproducing the errors present in them. Another important parameter is $x_6$ controlling the use of the *sharedVarNoise* heuristic. Moreover, the parameters $x_1$, $x_3$, and $x_6$ are considered important in all of

the rules, which suggests that, for reproducing the considered kind of errors, they are of a general importance.

In two cases, namely, *Crawler* and *Rover*, the *haltOneThread* heuristic ($x_4$) turns out to be relevant. In these test cases, the *haltOneThread* heuristic should be enabled in order to detect an error. This behaviour fits into our previous results [62] in which we show that, in some cases, this unique heuristic (the only heuristic which allows one to exercise thread interleavings which are normally far away from each other) considerably contributes to the detection of an error. Finally, the presence of the $x_{10}$ and $x_{12}$ parameters in the rules derived for the *Airlines* test case indicates that the number of threads ($x_{10}$) and the number of cycles executed during the test ($x_{12}$) pays an important role in the noise-based testing of this particular test case. The $x_{10}$ parameter (i.e. the number of threads) turns out to be important for the *Elevator* test case too, indicating that the number of threads is of a more general importance.

Finally, we can see that the $x_8$, $x_9$, and $x_{11}$ parameters are not present in any of the derived rules. This indicates that the *coverage-based* noise placement heuristics are of a low importance in general, and the $x_{11}$ parameter specific for *Airlines* is not really important for finding errors in this test case.

Next, for the case of classifying according to the rare behaviours property, the obtained rules are shown in Table 4.6. The highest weights can again be found in rules based on the $x_3$ parameter (*Animator*, *Crawler*, *Rover*, *Cache4j*, *HEDC*, *Montecarlo*, *Sor*, *TSP*) and on the $x_1$ parameter (*Airlines*). However, in the case of *Elevator* and *Raytracer*, the most contributing parameter is now the number of threads used by the test ($x_{10}$). Moreover, the $x_{10}$ parameter is also important in the *Montecarlo*, *Sor*, and *TSP* test cases. This suggests that choosing the right number of threads is quite important to maximize the chances to spot rare behaviours, and that it is not necessarily the case that the higher number of threads is used the better. Further, the generated sets of rules often contain the $x_3$ parameter controlling the type of noise (all test cases except for *Airlines* and *Raytracer*) and the $x_6$ parameter which controls the *sharedVarNoise* heuristic. These parameters thus appear to be of a general importance for the rare behaviours property.

The parameter $x_{12}$, i.e. the number of test cycles, does again turn out to be important in the *Airlines* test case. Finally, the $x_8$ parameter is shown only in one test case (*TSP*), $x_9$ shows up in the rules generated for two test cases (*Cache4j* and *TSP*), and the $x_{11}$ parameter does not show up in any of the rules, and hence seem to be of a low importance in general for finding rare behaviours (which is the same as for reproduction of known errors).

Overall, the obtained results confirmed some of the facts we discovered during our previous experimentation such as that different goals and different test cases may require a different setting of noise heuristics [62, 44, 42] and that the *haltOneThread* noise injection heuristics ($x_4$) provides in some cases a dramatic increase in the probability of spotting an error [62]. More importantly, the analysis revealed (in an automated way) some new knowledge as well. Mainly, the type of noise ($x_3$) and the setting of the *sharedVarNoise* heuristic ($x_6$) as well as the frequency of noise ($x_1$) are often the most important parameters (although the importance of $x_1$ seems to be a bit lower). Further, it appears to be important to suitably adjust the number of threads ($x_{10}$) whenever that is possible.

Table 4.6: Rules inferred for the rare behaviours property.

Airlines

| Rules | $\mathbf{x_1 \leq 295}$ or $745 < x_1 \leq 925$ | $x_2 \leq 35$ | $0.5 < x_5$ | $61.5 < x_{12} \leq 91.5$ |
|---|---|---|---|---|
| Weights | 0.52 | 0.06 | 0.1 | 0.32 |

Animator

| Rules | $\mathbf{0.5 < x_3 \leq 3.5}$ or $4.5 < x_3$ | $0.5 < x_6 \leq 1.5$ |
|---|---|---|
| Weights | 0.80 | 0.20 |

Crawler

| Rules | $\mathbf{0.5 < x_3 \leq 3.5}$ or $4.5 < x_3$ | $0.5 < x_4$ | $0.5 < x_5$ | $0.5 < x_6 \leq 1.5$ |
|---|---|---|---|---|
| Weights | 0.46 | 0.08 | 0.20 | 0.26 |

Elevator

| Rules | $0.5 < x_3 \leq 3.5$ or $\mathbf{4.5 < x_3}$ | $0.5 < x_4$ | $0.5 < x_5$ | $1.5 < x_6$ | $\mathbf{1.5 < x_{10} \leq 4.5}$ or $7.5 < x_{10}$ |
|---|---|---|---|---|---|
| Weights | 0.22 | 0.05 | 0.20 | 0.06 | 0.47 |

Rover

| Rules | $\mathbf{2.5 < x_3 \leq 3.5}$ or $4.5 < x_3$ | $x_4 \leq 0.5$ | $x_6 \leq 0.5$ | $0.5 < x_7$ |
|---|---|---|---|---|
| Weights | 0.46 | 0.26 | 0.16 | 0.12 |

Cache4j

| Rules | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$ | $x_5 \leq 0.5$ | $1.5 < x_6$ | $x_9 \leq 0.5$ |
|---|---|---|---|---|
| Weights | 0.92 | 0.02 | 0.05 | 0.01 |

HEDC

| Rules | $x_1 \leq 279$ | $49.5 < x_2$ | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$ | $1.5 < x_6$ |
|---|---|---|---|---|
| Weights | 0.03 | 0.02 | 0.89 | 0.06 |

Montecarlo

| Rules | $x_1 \leq 548.5$ | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3$ | $x_5 \leq 0.5$ | $0.5 < x_6$ | $x_9 \leq 0.5$ | $3.5 < x_{10} \leq 5.5$ |
|---|---|---|---|---|---|---|
| Weights | 0.09 | 0.30 | 0.05 | 0.18 | 0.09 | 0.29 |

Raytracer

| Rules | $20.5 < x_2 \leq 53.5$ or $\mathbf{75.5 < x_2}$ | $0.5 < x_5$ | $x_6 \leq 0.5$ | $0.5 < x_7$ | $x_{10} \leq 1.5$ or $\mathbf{4.5 < x_{10}}$ |
|---|---|---|---|---|---|
| Weights | 0.29 | 0.09 | 0.15 | 0.06 | 0.41 |

Sor

| Rules | $x_1 \leq 144.5$ | $x_3 \leq 1.5$ or $\mathbf{3.5 < x_3}$ | $0.5 < x_6$ | $x_7 \leq 0.5$ | $x_{10} < 13$ |
|---|---|---|---|---|---|
| Weights | 0.26 | 0.32 | 0.07 | 0.07 | 0.28 |

TSP – part1

| Rules | $x_1 \leq 691$ | $x_2 \leq 26$ | $\mathbf{x_3 \leq 0.5}$ or $3.5 < x_3 \leq 4.5$ | $x_5 \leq 0.5$ |
|---|---|---|---|---|
| Weights | 0.07 | 0.11 | 0.48 | 0.06 |

TSP – part2

| Rules | $0.5 < x_6$ | $0.5 < x_8$ | $x_9 \leq 0.5$ | $x_{10} \leq 18.5$ |
|---|---|---|---|---|
| Weights | 0.06 | 0.06 | 0.07 | 0.09 |

### 4.5.5 Fully-Automated Noise-based Testing with AdaBoost

We now present experimental results showing usefulness of the ways of applying AdaBoost in fully-automated noise-based testing that we proposed in Section 4.4.4. We consider both the combination of AdaBoost and random noise injection as well as the combination of AdaBoost and genetic algorithms. We start by considering the case of repeated reproduction of a known concurrency error and then proceed to the case of coverage of rare tasks.

**Repeated Error Manifestation**

Within our experiments aimed at repeated reproduction of known concurrency-related errors, we compare noise-based testing under test and noise configurations generated in the following ways:

- Purely random generation (referred to as *Random* below).
- Generation based on single-objective and multiple-objective genetic algorithms proposed in our earlier work and briefly described in chapter 3 (denoted as *SOGA* and *MOGA* below).
- Random generation filtered through the classic AdaBoost approach as described in the first part of Section 4.4.4 (referred to as *AdaBoost* in what follows).
- Random generation restricted to the AdaBoost-recognised most influential values of parameters described in the second half of Section 4.4.4 (denoted as *AdaBoost2* below).
- Generation based on the single-objective and multiple-objective genetic algorithms restricted to the AdaBoost-recognised most influential values of parameters as proposed in Section 4.4.4 (referred to as *SOGA2* and *MOGA2* below).

We run 5000 executions in the learning phase of those approaches that need some training. To compare capabilities of the obtained test and noise configurations in repeatadly finding the known errors, we then run 20 executions for 20 best configurations obtained through each of the approaches (apart from the random approach where we simply run 400 executions).

For experiments with the genetic algorithms, one has to choose the fitness function to be used. In particular, for the *SOGA* and *SOGA2* experiments, based on the experience we gained in our previous work, we have chosen the following fitness function:

$$fitness = \frac{Error}{Error_{max}} * 10 + \frac{Warning}{Warning_{max}} + \frac{GoldiLockSC^*}{GoldiLockSC^*_{max}} + \frac{time_{max} - time}{time_{max}}$$

Here, the *GoldiLockSC*$^*$ coverage metric is used since it has good properties for measuring general coverage of concurrency behaviour. The value *GoldiLockSC*$^*$ used in the fitness function gives the cumulative number of tasks covered in a series of five test runs performed with the given test and noise parameter values while $GoldiLockSC^*_{max}$ gives the maximal cumulative number of covered tasks across all so far performed series of test runs. However, since we want the fitness function to steer the search towards error discovery, we add to the fitness function information about the number of detected errors and error warnings. In particular, *Error* gives the number of error manifestations detected in the given series of five runs by looking for unhandled exceptions, and $Error_{max}$ gives the maximal number of error manifestations so far seen in some series of five test runs. $Warning$ gives the number of warnings detected in the given series of five test runs through the *Avio* checker [61] which detects atomicity violations over one variable. This metric has been chosen because atomicity violations are present in all the case studies considered in this experiment. Again, $Warning_{max}$ gives the maximum *Avio* coverage obtained in the so far performed series of test runs. Finally, as we want to reflect the time needed for the test runs, we add it into

the fitness function in such a way that lower amounts of time needed for the test runs are preferred[5].

For the *MOGA* and *MOGA2* experiments, we have let the multi-objective genetic algorithm work with the same objectives as those summarized in the fitness function of the *SOGA* and *SOGA2* approaches, i.e. the number of detected error manifestations, the *Avio* coverage, the *GoldiLockSC** coverage, and the needed testing time. In all our experiments with the genetic algorithms, we used the following settings: the probability of mutation was set to 0.5, the number of individuals in one population was 20, and each individual was evaluated by using the cumulative value from five executions of one configuration. We used the two-point crossover and the tournament selection operator (which provided us with the best results in our previous work in chapter 3). For each case study, we repeat each experiment ten times.

Table 4.7 compares results obtained using the above described approaches. In particular, the table presents numbers and percentages of the executions that managed to find an error in those of our benchmark programs that contain a known error. As we can see, the single-objective genetic algorithm restricted to the AdaBoost-selected most influential parameter values (i.e. *SOGA2*) has achieved the best results on average. However, random generation of test and noise parameter values restricted to the AdaBoost-selected most influential parameter values (*AdaBoost2*) and the combination of the multi-objective genetic algorithm and AdaBoost (*MOGA2*) have also achieved very good results.

Table 4.7: An experimental comparison of various fully-automated approaches to noise-based testing in the context of reproducing a known error. The best results are highlighted in bold.

| *CaseStudies* | *Random* error/ % | *SOGA* error / % | *MOGA* error/ % | *AdaBoost* error/ % |
|---|---|---|---|---|
| Airlines | 132.93/33.23 | 313.25/78.31 | 272.25/68.06 | 323.50/80.88 |
| Animator | 106.75/26.69 | 220.20/55.05 | 131.00/32.75 | 144.80/36.20 |
| Crawler | 0.00/0.00 | 0.50/0.13 | 0.50/0.13 | 0.80/0.20 |
| Elevator | 59.25/14.81 | **133.25/33.31** | 116.75/29.19 | 80.40/20.10 |
| Rover | 17.00/4.25 | 143.00/35.75 | 88.25/22.06 | 57.40/14.35 |
| Average | /15.80 | /40.51 | /30.44 | /19.11 |
| ASD | /6.01 | /5.50 | /7.91 | /7.44 |

| *CaseStudies* | *AdaBoost2* error/ % | *SOGA2* error/ % | *MOGA2* error/ % |
|---|---|---|---|
| Airlines | 351.80/87.95 | **371.80/92.95** | 332.7/83.13 |
| Animator | 252.40/63.10 | **350.30/87.58** | 241.25/60.31 |
| Crawler | 1.00/0.25 | **2.40/0.60** | 0.80/0.20 |
| Elevator | 36.60/9.15 | 105.00/26.25 | 86.80/21.70 |
| Rover | 48.4/12.65 | **324.80/81.20** | 203.30/50.83 |
| Average | /34.62 | /**57.72** | /43.24 |
| ASD | /4.91 | /4.89 | /**2.58** |

[5]Here, one could be tempted to divide the fitness values by the time needed. We do not use this approach since our previous experience presented in Chapter 3 showed that this often leads to significant degeneration of the search (producing configurations that produce very low coverage in extremely short time).

It must be noted that 14 generations were used for the *SOGA* and *MOGA* experiments, and 7 generations were used for the *SOGA2* and *MOGA2* experiments, which are very small numbers only. The reason for using such small numbers of generations is that we wanted to compare the different approaches while giving them the same time for the learning phase. The *MOGA2* approach had the lowest standard deviation on average. This means that the *MOGA2* approach gives good results with a high probability.

**Coverage of Rare Concurrent Behaviours**

Table 4.8: A comparison of average cumulative numbers of rare tasks over the time needed to cover them.

| *CaseStudies* | Rand. rareTasks/ % | SOGA rareTasks/ % | MOGA rareTasks/ % | AdaBoost rareTasks/ % |
|---|---|---|---|---|
| Airlines | 0.6566/ 41.4 | 1.2950/ 81.6 | 1.5462/ 97.4 | 0.4768/ 30.0 |
| Animator | 7.0193/ 4.6 | 145.8694/ 95.3 | **153.0821/ 100.0** | 87.3576/ 57.1 |
| Cache4j | 0.0165/ 38.9 | 0.0167/ 39.4 | 0.0413/ 97.4 | 0.0292/ 68.9 |
| Crawler | 3.0415/ 51.1 | 4.7546/ 79.9 | 3.1230/ 52.5 | 3.6581/ 61.5 |
| Elevator | 9.0015/ 48.1 | 13.5446/ 72.4 | 16.9801/ 90.8 | 17.4073/ 93.1 |
| HEDC | 0.3605/ 22.1 | 0.9909/ 60.7 | 0.7595/ 46.5 | 0.9754/ 59.7 |
| Montecarlo | 0.1469/ 59.9 | 0.2158/ 88.0 | **0.2453/ 100.0** | 0.1482/ 60.4 |
| Raytracer | 0.0009/ 7.7 | 0.0003/ 2.6 | 0.0003/ 2.6 | 0.0006/ 5.1 |
| Rover | 1.1532/ 42.1 | 1.7713/ 64.6 | 1.5623/ 57.0 | 1.4008/ 51.1 |
| Sor | 0.0497/ 25.4 | 0.0742/ 37.9 | 0.0860/ 44.0 | 0.1088/ 55.6 |
| TSP | 0.0381/ 36.9 | 0.0659/ 63.9 | 0.0971/ 94.1 | 0.0520/ 50.4 |
| Average | / 34.4 | / 62.4 | / 71.1 | / 55.6 |
| ASD | / 17.6 | / 26.9 | / 32.5 | / 20.7 |

| *CaseStudies* | AdaBoost2 rareTasks/ % | SOGA2 rareTasks/ % | MOGA2 rareTasks/ % |
|---|---|---|---|
| Airlines | 0.9298/ 58.6 | **1.5876/ 100.0** | 1.1216/ 70.6 |
| Animator | 136.5519/ 89.2 | 114.9578/ 75.1 | 110.4470/ 72.1 |
| Cache4j | 0.0194/ 45.8 | 0.0389/ 91.7 | **0.0424/ 100.0** |
| Crawler | 5.8669/ 98.6 | 4.1439/ 69.6 | **5.9502/ 100.0** |
| Elevator | **18.7019/ 100.0** | 14.9516/ 79.9 | 17.1540/ 91.7 |
| HEDC | 1.1568/ 70.8 | 1.3836/ 84.7 | **1.6334/ 100.0** |
| Montecarlo | 0.1780/ 72.5 | 0.1664/ 67.8 | 0.1823/ 74.3 |
| Raytracer | 0.0052/ 44.4 | **0.0117/ 100.0** | 0.0104/ 88.9 |
| Rover | 1.3018/ 47.5 | 1.9877/ 72.5 | **2.7411/ 100.0** |
| Sor | 0.1154/ 59.0 | 0.1855/ 94.8 | **0.1956/ 100.0** |
| TSP | 0.0642/ 62.2 | 0.0867/ 84.0 | **0.1032/ 100.0** |
| Average | / 67.7 | / 83.6 | / **90.7** |
| ASD | / 20.5 | / **11.8** | / 12.4 |

In the second part of our experiments, we concentrate on increasing coverage of rare concurrent behaviours. Compared with the experiments of the previous section, we consider all of our benchmark programs since we do not need them to contain an error. For the *SOGA*

and *SOGA2* approaches, we use the following simplified fitness function:

$$fitness = \frac{GoldiLockSC^*}{GoldiLockSC^*_{max}} + \frac{time_{max} - time}{time_{max}}.$$

From the fitness function, we have left out information about errors and warnings since we now do not focus on occurrences of any known errors. The *MOGA* and *MOGA2* approaches are based on the same objectives as *SOGA* and *SOGA2*, i.e. *time* and $GoldiLockSC^*$. As in the experiments of the previous section, the probability of mutation was set to 0.5, and each individual was evaluated using cumulative coverage obtained in five runs. Each generation had 20 individuals.

For the random approach, we executed 1000 test runs with randomly generated test and noise configurations. For the other approaches, we used the same number of test runs, which we divided into 500 runs to train the approaches and the remaining 500 runs to execute the test cases with the configurations obtained from the training phase. When training the AdaBoost-based approaches, we took as positive (i.e. suitable for testing) 50 configurations with the highest results of cumulative coverage obtained from five runs and the other configurations as negative. For the approaches based purely on genetic algorithms, i.e. *SOGA* and *MOGA*, we used five generations in the training phase. For the combination of AdaBoost and genetic algorithms, i.e. *SOGA2* and *MOGA2*, we used 250 runs for training AdaBoost and three generations for the subsequent training of the genetic algorithms. For each case study, we repeated each experiment ten times.

In Table 4.8, we present results of the above experiments (which took in total approximately 6,939 core hours, i.e. 289 core days). In particular, the entries of the table contain—for the different programs and different approaches—the obtained coverage of rare tasks over the time needed to obtain the coverage. We divide the obtained coverage by the needed time in order to better see which of the approaches is better to quickly obtain a high coverage of rare tasks. Moreover, the obtained coverage over the testing time is followed by its interpretation in per cent. Namely, the approach with one hundred per cent is the winning one, and, for the others, the percentage shows how far they are from the winning approach in terms of the achieved coverage over time. As we can see, the combinations of AdaBoost with the genetic approaches (i.e. *MOGA2* and *SOGA2*) have the best results on average, and they are also more stable than the other methods.

## 4.6 Conclusions and Future Work

In this chapter, we have proposed a novel application of data mining in the context of noise-based testing of concurrent programs. In particular, we have employed data mining based on binary classification, decision trees, and the AdaBoost machine learning algorithm. We have shown how to use these technologies for finding a suitable set up of noise injection, i.e. selecting suitable noise injection heuristics out of the many known ones and finding suitable values of their various parameters, with the aim of maximizing chances of meeting a given testing goal. We have illustrated our approach on two concrete testing goals in the context of concurrent programs, namely, reproduction of known errors for debugging purposes and covering rare behaviours, which are more likely to contain so far unknown bugs than common behaviours. We have shown how data mining can be used to gain more insight into the suitability of the different noise heuristics and their parameters, allowing testers to choose the right ones for the given context, as well as how to use data mining to improve fully automated noise-based testing. For the latter case, we have combined

our approach both with noise-based testing on a random basis as well as with genetically optimized noise-based testing. For all the proposed approaches, we have illustrated on a number of case studies that they can indeed improve the process of noise-based testing of concurrent programs.

In the future, we would like to apply in the context of testing of concurrent programs other approaches to data mining than AdaBoost and binary classification that we considered in this chapter. This could include approaches such as outliers detection, clustering, or association rules mining. We would also like to look for other applications of data mining than setting up noise injection in a suitable way. For example, many of the concurrency coverage metrics based on dynamic detectors contain a lot of information on the behaviour of the tested programs, and when mined, this information could be used for debugging purposes. One could also think of generalising the various existing works devoted to detection of untested behaviour or to eliminating tests of similar behaviour of sequential programs (cf. Section 4.2) for the case of concurrent programs.

# Chapter 5

# Prediction Coverage of Expensive Metrics from Cheaper Ones

We already know from previous chapters that analysing of concurrent programs is very difficult due to scheduling non-determinism. To find suitable values of test and noise parameters, when one uses noise-based testing or analysis. For maximizing coverage under some metrics, one may need a large number of test executions, which is time-consuming. To minimize this problem, we show that there are correlations between metrics of different cost and that one can find a suitable test and noise setting to maximize coverage under costly metrics by experiments with cheaper metrics.

## 5.1 Introduction

To maximize coverage under a chosen concurrency coverage metric (or a combinations of such metrics), the space of possible thread schedules has to be properly examined. If the TNCS problem is not solved properly, the usage of noise can even decrease the obtained coverage [30]. However, solving the TNCS problem is not an easy task. Sometimes, its solution is not even attempted, and purely random noise generation is used. Alternatively, one can use genetic algorithms or data mining [42, 44, 6]. These approaches can outperform the purely random approach, but finding suitable test and noise settings this way can be quite costly. The aim of this chapter is to make the cost of this process cheaper.

The approach which we propose builds on the facts that (1) maximizing coverage under different metrics may have different *costs*, and that (2) one can find *correlations* between test and noise settings that are suitable for maximizing coverage under different metrics. Moreover, such correlations may link even metrics for which the process of maximizing coverage is expensive but which are highly informative for steering the testing process and metrics for which the process of maximizing coverage is cheaper but which are less efficient when used for steering the testing process. We confirm all these facts through a set of our experiments. In particular, we identify the correlations by building a *predictive model* between several expensive metrics (under which one may want to simultaneously maximize coverage) and several cheap metrics.

Using the above facts, we suggest to *optimize the testing process* in the following way. Given some expensive but informative metrics, one may find suitable values of test and noise parameters for maximizing coverage under these metrics by experimenting with coverage under some cheap metric (or a combination of such metrics) and then use this setting for

testing with the expensive metrics. We show on a set of experiments that this approach can indeed increase the efficiency of noise-based testing.

Our contribution is thus threefold: (1) An experimental categorisation of various concurrency-related metrics to cheap and expensive ones according to the price of maximizing coverage under these metrics. (2) The observation and experimental confirmation of correlations between test and noise settings suitable for testing under metrics of different cost. (3) The idea of exploiting the above facts for more efficient noise-based testing of concurrent programs and its experimental evaluation.

## 5.2 Related Work

In previous chapters, we focused on solving the test and noise problem via genetic algorithms and data mining. Here, we propose an orthogonal optimisation based on solving the TNCS problem for expensive concurrency metrics by using cheaper ones, which is justified by existence of a predictive model between the expensive and cheap metrics. Prediction is used in various other areas of software testing, e.g. to predict bug severity [65] or to link concurrency-related code revisions with the corresponding issues and characterize bugs [18]. None of these works, however, builds on prediction in a similar way as our work in this chapter.

## 5.3 Preliminaries

In this section, we briefly introduce regression methods, as well as the benchmark programs and experimental setting used in the rest of the chapter.

### 5.3.1 Regression Models

In the following part of the chapter, we briefly introduce three algorithms which are mostly used to create regression models. These three algorithms are stepwise regression, ridge regression, and the LASSO algorithm. We discuss their usage in the context below and conclude that the LASSO algorithm suits as the best.

Our motivation for using some regression algorithm is to find a combination of cheap metrics whose coverage could predict some expensive metrics. For our purpose, we also need to select the ideal number of cheap metrics which is necessary for creation of the prediction model.

For the regression models, suppose that we have data $(\mathbf{x}^i, y_i)$, $i = 1, 2, \ldots, N$ where $\mathbf{x}^i = (x_{i1}, \ldots, x_{ip})^T$ are the predictor variables (cheap metrics) and $y_i$ are the responses (expensive metrics). As is usual in regression, we assume either that the observations are independent or that the $y_i$s are conditionally independent given the $x_{ij}$s.

#### Stepwise Regression

Stepwise regression is a classical statistical method which calculates the F-value for incremental inclusion of each variable in the regression. The F-value is an equivalent to the square root of the Student's t-value, expressing how different two samples are from each other. The t-value is calculated as

$$t = \frac{\overline{X} - \mu}{\sqrt{\frac{s^2}{n}}},$$

which represents the difference between a sample mean (i.e. average) $\overline{X}$ and the population mean $\mu$ divided by the standard deviation of the sample $s$, and so

$$F = \sqrt{t\text{-value}}.$$

The F-value is sensitive to the number of variables used for its calculation. Stepwise regression calculates the F-value both with and without using a particular variable and compares it with a critical F-value either to include the variable (*forward stepwise selection*) or to eliminate the variable from the regression (*backward stepwise selection*) [39]. This algorithm can be used to select the variables which are in our case cheap metrics.

### Ridge Regression

The most popular form of regularized regression is *ridge regression*, which places a constraint on the sum of squares of the coefficient's weights. Formally, ridge regression perfects the residual (Error) sum of squares (RSS) subject to a constraint on P — in our case, this means the number of cheap metrics used for prediction. Ridge regression is motivated by a constrained minimization problem, which can be formulated as follows:

$$\beta_{ridge} = argmin_{\beta \in \mathbf{R}^p} \sum_{i=1}^{n} (Y_i - X_i^T \beta)^2 \text{ subject to } \sum_{j=1}^{p} \beta_j^2 \leq t$$

for $t \geq 0$ which is a so-called tuning parameter. Moreover, the coefficient $\beta_0$ is excluded from the penalty term [39, 36].

### The LASSO Algorithm

The *LASSO* (least absolute shrinkage and selection operator) algorithm, by contrast to ridge regression, tries to produce a sparse solution, in the sense that several of the slope parameters will be set to zero. One may therefore refer to ridge regression as soft thresholding, whereas the LASSO algorithm is soft/hard, and the subset selection is a hard thresholding; since, in the latter, only a subset of the variables is included in the final model.

As in ridge regression, the LASSO algorithm can be expressed as a constrained minimization problem by the following equation:

$$\beta_{LASSO} = argmin_{\beta \in \mathbf{R}^p} \sum_{i=1}^{n} (Y_i - X_i^T \beta)^2 \text{ subject to } \sum_{j=1}^{p} |\beta_j| \leq t$$

where $t \geq 0$ is a tuning parameter.

Generally, computing the LASSO algorithm solution is a quadratic programming problem. A small enough $t$ will set some coefficients exactly equal to 0. Thus, the LASSO algorithm does a kind of continuous subset selection. Like the subset size in variable subset selection, or the penalty parameter in ridge regression [39], $t$ should be adaptively chosen to minimize the estimated prediction error.

### Comparison of the Regression Methods

Statistical methods which are introduced in the previous paragraphs are used mostly in other scientific disciplines than information technology and testing of concurrent programs. Such disciplines are, for example, biology, meteorology, etc. This is the reason why this subsection presents papers from other disciplines.

In [77], they compare stepwise algorithms with some alternative approaches such as the LASSO algorithm. The paper says that although the stepwise algorithms remain the dominant method in some part of science, the automatic stepwise subset selection methods often perform poorly, both in terms of variable selection and estimation of coefficients and standard errors, especially when the number of independent variables is large and multicollinearity is present. The use of stepwise methods were outperformed by alternative methods.

Moreover, paper [100] describes the procedure of stepwise regression and uses experiments and Venn diagrams to illustrate the three main problems of stepwise regression: a wrong degree of freedom, capitalization on sampling, and the $R^2$ error not optimized.

In [35], they use a different downscaling statistical methods for prediction where between them is also LASSO regression. The LASSO algorithm was tested and validated against three other downscaling methods, namely, the local intensity method, quantile-mapping, and stepwise regression. Compared to these three downscaling methods, LASSO algorithm shows the best performances. Furthermore, LASSO algorithm could reduce the error for certain sites, where no improvement could be seen when other methods were used. The study proves that LASSO is a reasonable alternative to other statistical methods with respect to the downscaling of precipitation data.

In [8] the authors compared linear regression with the regularized regressions such as ridge and LASSO regressions because multicollinearity is one of the major problems in regression analysis, and it could be reduced by using regularized regressions. They find that, in every considered data set, LASSO and ridge models have smaller RSS[1] value, and they conclude that regularized models are best fitting models in regression analysis when one found noise exists in the usual models.

A conclusion of the comparison of these three regression methods is that ridge and LASSO algorithms are better than stepwise regression. Moreover for our purposes, we need a method with the variable selection, which is the LASSO algorithm. Thus, we chose the LASSO algorithm in our approach for prediction of expensive metrics from cheaper ones.

## 5.3.2 Benchmarks and Experimental Setting

The experimental results presented below are based on the following 10 multithreaded benchmark programs written in Java: *Airlines* (0.3 kLOC), *Cache4j* (1.7 kLOC), *Animator* (1.5 kLOC), *Crawler* (1.2 kLOC), *Elevator* (0.5 kLOC), *HEDC* (12.7 kLOC), *Montecarlo* (1.4 kLOC), *Rover* (5.4 kLOC), *Sor* (7.2 kLOC) and *TSP* (0.4 kLOC). More details about these benchmarks can be found in Section 2.6. All our experiments were performed using the IBM ConTest tool [23] on a machine with Intel Xeon E3-1240 v3 processors at 3.40GHz, 32GiB RAM, running Linux Debian 3.16.36, and using OpenJDK version 1.8.0_111.

[1]The RSS value means the residual sum of squares.

## 5.4 Increasing Coverage of Expensive Metrics by Prediction

In the following section, we introduce an approach how to predict coverage of multiple expensive metrics using a prediction model based on cheap metrics. First, we focus on a classification of the metrics cost. Then, we create the prediction model based on the cheap metrics, and, finally, we execute and evaluate experiments with the model.

### 5.4.1 Distinguishing Cheap and Expensive Metrics

We now explain our way of distinguishing cheap and expensive metrics, i.e. metrics for which collecting coverage is cheaper or more expensive, respectively.

For the classification of the cost of the metrics, we first ran a series of 1000 test runs of each of our benchmark programs without collecting any coverage. These tests were, however, run already in the ConTest environment, using its random noise setting, which already slows the programs down. This way, we obtained the so-called *bottom case*. The running time of the tests in the bottom case was around 93 seconds for one execution when averaging over all our case studies.

Second, for each metric, we performed 100 test runs while collecting coverage under the given metric, again using ConTest with random noise injection. We then compared the time needed for the bottom case with the times of the experiments with each single metric. We classify metrics into three groups: cheap metrics, expensive metrics, and others (i.e. metrics with medium slowdown). In particular, we mark metrics with the slowdown between 10 % and 30 % as cheap metrics and those with the slowdown 50 % and more as expensive metrics.

### 5.4.2 Discovering Correlations between Cheap and Expensive Metrics

Next, we aim at automatically finding correlations between metrics that will allow us to find suitable test and noise settings for testing under expensive metrics by experimenting with cheaper ones. Due to multiple metrics are often used in testing of concurrent programs (each of them stressing somewhat different aspects of the behaviour), we, in fact, aim at correlations between sets of expensive metrics and sets of cheap metrics.

For the above, one can use *multi-variable regression* on the cumulative coverage of the different metrics obtained from multiple test runs (i.e. coverage based on a union of the sets of coverage tasks covered in the different runs). However, we, instead, decided to use one from the regression method presented in the preliminaries section. For our goals, we chose the regression methods which include selection of variables because from the set of cheap metrics we need to choose a subset of metrics for prediction. Based of the comparison of the three common regression methods presented in 5.3.1, we use for our experiments the LASSO algorithm [49, 39] to build a *predictive model* between cheap and expensive metrics. The algorithm selects suitable cheap metrics and constructs their linear combination capable of predicting a given expensive metric, hence showing correlation among the metrics. In our experience, this approach gives more stable results than normal correlation.

In more detail, we use the LASSO algorithm to search for a combination of cheap metrics which has a high partial correlation coefficient with a chosen expensive metric. The algorithm iteratively increases the partial correlation and selects a subset of cheap metrics with

the highest partial correlation. The obtained predictive model then looks as follows:

$$expMetric = \beta_0 + \beta_1 * cheapMetric^1 + \cdots + \beta_n * cheapMetric^n.$$

Note that the above model predicts a single expensive metric based on several cheap ones. However, we aim at maximizing the coverage under *several* expensive metrics based on the settings suitable for several cheap metrics. To handle this, we propose to replace the role of the single expensive metric in the above model by using a *fitness function* representing a weighted combination of the chosen expensive metrics (as often done in genetic algorithms).

Such a combination can have the following form:

$$fitness = \frac{expMetric^1}{expMetric^1_{max}} + \cdots + \frac{expMetric^n}{expMetric^n_{max}}.$$

Here, $expMetric^i$ is the cumulative coverage under the $i$-th metric obtained in the given series of test runs with the same test and noise setting, while $expMetric^i_{max}$ is the maximum of all cumulative coverage values under the given metric in all experiments performed so far, even with different test and noise settings. This way way of approximating the maximum is used, since there is no exact way of computing it.

## 5.5 Experimental Results

In this section, we present the individual results of our the experiments in the following order: classification of the metrics according to their the slowdown incurred when collecting coverage under these metrics; the regression model for prediction of the expensive metrics using the cheap ones; and the approach to the testing of concurrent programs and increasing the coverage of the expensive metrics.

### 5.5.1 Results of Metric Costs Classification

We divide the metrics into three classes. The cheapest metrics have the slowdown between 10 % and 30 %, the most expensive metrics have the slowdown of 50 % and more. The rest are metrics with a medium slowdown. As mentioned in Section 5.4.1, the slowdown was obtained by comparing the time needed to perform 100 test runs while collecting coverage under the different metrics against the time needed to run the test runs under ConTest but without collecting coverage. The obtained classification is shown in Table 5.1 and used in the further experiments.

### 5.5.2 Regression Model for Prediction

We decided to experiment with finding suitable test and noise settings by simultaneously maximizing the coverage under all the three identified expensive metrics: *GoldiLockSC**, *WEraser**, and *Datarace*. The first step was to construct a fitness function combining these three metrics for using the LASSO algorithm. For this purpose, we generated 100 random test and noise settings, ran five tests with each configuration, cumulating the coverage obtained in these runs. Then, we took the maximum values of the cumulated coverage from the 100 experiments. We obtained the following fitness function:

$$fitness = \frac{GoldiLockSC^*}{1443} + \frac{WEraser^*}{3862} + \frac{Datarace}{267}.$$

Table 5.1: Cheap and expensive metrics.

| | ***Slowdown in %*** | ***Metrics*** |
|---|---|---|
| **Cheap metrics** | $10\,\% \leq x < 30\,\%$ | Avio, Avio*, Concurpairs, HBPair, GoodLock, ShvarPair*, Synchro, WSynchro |
| **Medium slowdown metrics** | $30\,\% \leq x < 50\,\%$ | Deadlock, SharedVar, GoodLock*, ShvarPair, WConcurpairs, Eraser, Eraser*, Dupair, Dupair*, Atomviolat, LockSet, HBPair* |
| **Expensive metrics** | $50\,\% \leq x$ | Datarace, WEraser*, GoldiLockSC*, GoldiLock, GoldiLock* |

Secondly, we used the LASSO algorithm with forward regression as implemented in the *glmnet()* function from the *glmnet* package [49] of the R-project tool to obtain the predictive model. We created the predictive model from a cumulation of results from the five runs on all the considered case studies.

In the forward LASSO algorithm, it is possible to choose the number of cheap metrics for the prediction. This is because the algorithm starts with an empty model and in each step, it adds one cheap metric to the previously built prediction model. Thus, we can see which cheap metrics form the model in each iteration. For our case, we chose to predict three expensive metrics by only two cheap metrics. In the second part of this chapter, we focus on the comparison of the prediction using two, three, or four cheap metrics. We assume that using more cheap metrics for the prediction could be more precise, but also more time-consuming.

Using the above approach, we obtained the following predictive model:

$$fitness = 2.9e-01 + 2.2e-06 * ConcurPairs + 1.8e-03 * Avio^{*}.$$

This predictive model and also the aforementioned fitness function are used in all further experiments described in the next sections.

### 5.5.3 Using Correlations of Metrics to Optimize Noise-based Testing

Once the predictive model is created and we know which set of cheaper metrics can be used to predict the coverage under a given (set of) expensive metrics, this knowledge can be used to optimize the noise-based testing process. In particular, we can try to find suitable test and noise settings for the given expensive metrics by experimenting with the cheap ones. The experiments can be controlled using a genetic algorithm [42, 44] or by data mining on the test results [6], all the time evaluating the performed experiments via the chosen cheap metrics or, more precisely, through the predictive model built. In the simplest case, only a number of random experiments with different test and noise settings can be performed. Then the settings that performed the best in these experiments wrt. the predictive model are chosen. This is the approach we follow below to show that our approach can indeed improve the noise-based testing process.

We randomly generated 100 test and noise configurations and executed five test runs with each of them for each one of our case studies, while collecting the coverage under the selected cheap metrics (this led to 500 executions for each case study). We cumulated the results within the five executions of one configuration and then worked with the obtained cumulative value. We chose 20 configurations with the best results wrt. the derived predictive model. These 20 configurations were used for further test runs under the three considered expensive metrics. Each of the chosen 20 configurations was executed 200 times, leading to 4000 test executions under the three expensive metrics for each case study. Finally, to compare the efficiency of this approach with the purely random one, we also performed 4500 test runs with random test and noise settings while directly collecting the coverage under the expensive metrics for each one of the case studies. Hence, both of the approaches were given the same number of test runs.

Table 5.2: A comparison of random and prediction-optimized noise-based testing.

| | GoldiLockSC* | | WEraser* | | Datarace | |
|---|---|---|---|---|---|---|
| *CaseStudies* | Random | Predict | Random | Predict | Random | Predict |
| Airlines | 9.46 | **22.42** | 74.92 | **182.59** | 0.28 | **0.72** |
| Animator | 817.82 | **1451.35** | 233.20 | **291.42** | 0.35 | **0.46** |
| Cache4j | 0.93 | **2.62** | 4.14 | **10.98** | 0.03 | **0.10** |
| Crawler | 54.93 | **88.69** | 351.85 | **547.41** | 1.90 | **2.86** |
| Elevator | **297.09** | 286.30 | **756.72** | 733.91 | **2.31** | 2.23 |
| HEDC | **27.50** | 19.93 | **67.37** | 48.73 | **0.50** | 0.36 |
| Montecarlo | 4.24 | **5.19** | 9.03 | **11.35** | 0.02 | **0.03** |
| Rover | 37.62 | **62.89** | 174.14 | **292.18** | **0.08** | **0.08** |
| Sor | 3.19 | **7.16** | 4.93 | **12.69** | **0.00** | **0.00** |
| TSP | **1.86** | 1.40 | **15.36** | 11.74 | **1.14** | 0.86 |
| Average Impr. | | **1.62** | | **1.59** | | **1.46** |

In Table 5.2, we compare the random approach with our prediction-based approach. In particular, we aim at checking whether the proposed approach can help to increase the obtained coverage of the expensive metrics when weighted by the consumed testing time. We can see in the table that this is indeed the case: the coverage over time increased in most of the cases. The average improvement of the obtained cumulative coverage over the testing time across all our case studies ranges from 46 % to 62 %.

Figure 5.1 (right) compares how the obtained cumulative coverage, averaged over all of our case studies, grows when increasing the number of performed test runs under the purely random noise-based approach and under our optimized approach. Our approach has better results, despite having an initial penalty because of the use of a number of test runs to find suitable test and noise parameters via cheap metrics. The left part of the figure then compares the average time needed by the two approaches over all the case studies. Again, the optimized approach shows better results.

## 5.6 Discovering Ideal Number of Cheap Metrics to Increase Performance

In the next experiments, we want to predict any three given metrics: not only the expensive one, but possibly the cheap ones as well. We present three experiments, in which we try to

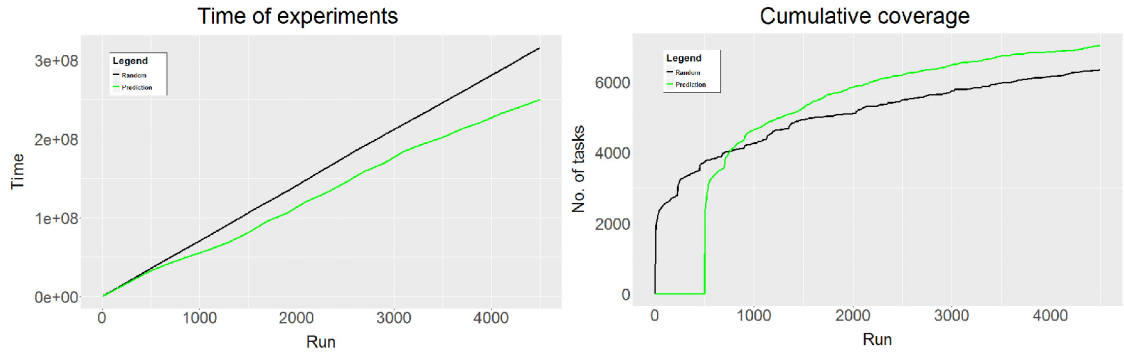



Figure 5.1: Testing time (left) and cumulative coverage (right) for an increasing number of test runs.

predict the given fitness function by two, three, and four cheap metrics, and we compare the results. Such experiments show us the ideal number of cheap metrics to predict some other metrics (at least for the cases considered so far). The process of experiments is the same as in the previous part of this chapter.

### 5.6.1 Results of Creation Predictive Models

We decided to experiment with finding suitable test and noise settings by simultaneously maximizing the coverage under the three given metrics (*GoldiLockSC**, *WConcurPairs*, and *HBPair**) and by minimizing the running time (i.e. we have four metrics). The first step was to construct a fitness function combining these four metrics by the LASSO algorithm. for this purpose, we followed the same procedure as in the previous experiments: the same setting of the number of runs, finding the maximal coverage, etc. This way, we obtained the following fitness function:

$$fitness = \frac{GoldiLockSC^*}{1443} + \frac{WConcurPairs}{2811899} + \frac{HBPair^*}{120} + \frac{3504127 - time}{3504127}.$$

As in the previous experiment with three expensive metrics, we used the LASSO algorithm with forward regression to obtain the predictive model. In particular, we aimed at predicting the four given metrics by two, three, and four cheap metrics. We assumed that using more cheap metrics for the prediction could be more precise, but also more time-consuming.

Using the above approach, we obtained the following predictive models. For two cheap metrics:

$$model1 = 1.039 + 0.012 * SYNCHRO + 0.00076 * SHVARPAIRT;$$

for three cheap metrics:

$$model2 = 1.0345 + 0.0118 * SYNCHRO + 0.0011 * SHVARPAIRT$$
$$-0.00035 * AVIOTRIPT;$$

and for four cheap metrics:

$$model3 = 1.02 + 0.016 * SYNCHRO + 0.00165 * SHVARPAIRT$$

$$-0.00087 * AVIOTRIPT - 7.27e^{-06} * WSYNCHRO.$$

These predictive models and also the aforementioned fitness function are used in all further experiments described below.

The correlations of the fitness function and the combinations of cheap metrics are as follows: for two cheap metrics, the correlation is 0.8470027; for three cheap metrics, it is 0.8510919; and for four cheap metrics, it is 0.8559141. The correlation of the given fitness function and the combination of cheap metrics is very high for each considered types of prediction. The correlations are also very close to each other, which may indicate that there is no big difference between them.

### 5.6.2 Results of Models Comparison

Table 5.3: A comparison of random noise based tests and three prediction-optimized settings of noise-based testing.

| | | GoldiLockSC* | | |
|---|---|---|---|---|
| *CaseStudies* | Rand. | model1 | model2 | model3 |
| Airlines | 0.36 | **1.01** | 0.60 | 0.59 |
| Cache4j | 0.43 | **0.95** | 0.62 | 0.65 |
| Crawler | 32.15 | 73.52 | **76.94** | 75.59 |
| Elevator | 31.65 | **36.59** | 36.44 | 35.63 |
| HEDC | 55.65 | 38.00 | 53.77 | **58.68** |
| Rover | 35.61 | **61.24** | 48.24 | 47.51 |
| Average Impr. | | **1.74** | 1.48 | 1.48 |

| | | WConcurPairs | | |
|---|---|---|---|---|
| *CaseStudies* | Rand. | model1 | model2 | model3 |
| Airlines | 0.0010 | **0.0020** | 0.0012 | 0.0012 |
| Cache4j | 0.0000 | **0.0001** | 0.0000 | 0.0000 |
| Crawler | 0.0130 | 0.0265 | **0.0281** | 0.0272 |
| Elevator | 0.0101 | **0.0102** | 0.0100 | 0.0100 |
| HEDC | 0.0006 | 0.0004 | 0.0006 | **0.0007** |
| Rover | 0.0021 | **0.0036** | 0.0028 | 0.0029 |
| Average Impr. | | **1.54** | 1.37 | 1.39 |

| | | HBPair* | | |
|---|---|---|---|---|
| *CaseStudies* | Rand. | model1 | model2 | model3 |
| Airlines | 0.0038 | **0.0078** | 0.0044 | 0.0049 |
| Cache4j | 0.0003 | **0.0006** | 0.0004 | 0.0004 |
| Crawler | 0.0726 | 0.1683 | **0.1757** | 0.1731 |
| Elevator | **0.1004** | 0.0937 | 0.0906 | 0.0901 |
| HEDC | 0.0130 | 0.0095 | 0.0131 | **0.0141** |
| Rover | 0.0272 | **0.0371** | 0.0306 | 0.0325 |
| Average Impr. | | **1.55** | 1.35 | 1.38 |

As in the case of the three expensive metrics, we randomly generated 100 test and noise configurations and executed five test runs with each of them for each one of our case studies, while collecting the coverage under the selected cheap metrics (leading to 500 executions for each case study). We cumulated the results within the five executions of one configuration and then worked with the obtained cumulative value. We chose 20 configurations with the best results wrt. the derived predictive model. These 20 configurations were used for further test runs under the four considered metrics. Each of the chosen 20 configurations was executed 200 times, leading to 4000 test executions under the four given metrics for each case study. Finally, to compare the efficiency of this approach with the purely random one, we also performed 4500 test runs with random test and noise settings, while directly collecting the coverage under the four given metrics for each one of the case studies. Hence, both of the approaches were given the same number of test runs.

In Table 5.3, we compared the random approach with three prediction-based approaches. From the previous experiment with three expensive metrics, we know that the prediction optimization works relatively well. Now, we wanted to find how many cheap metrics must be used for prediction for the best results. In the table, we can see that the results between model1, model2 and model3 are not very different, but the improvement is the highest in the prediction with two cheap metrics. The average improvement of the obtained cumulative coverage over the testing time across all our case studies is more than 50 % in the case of model1.

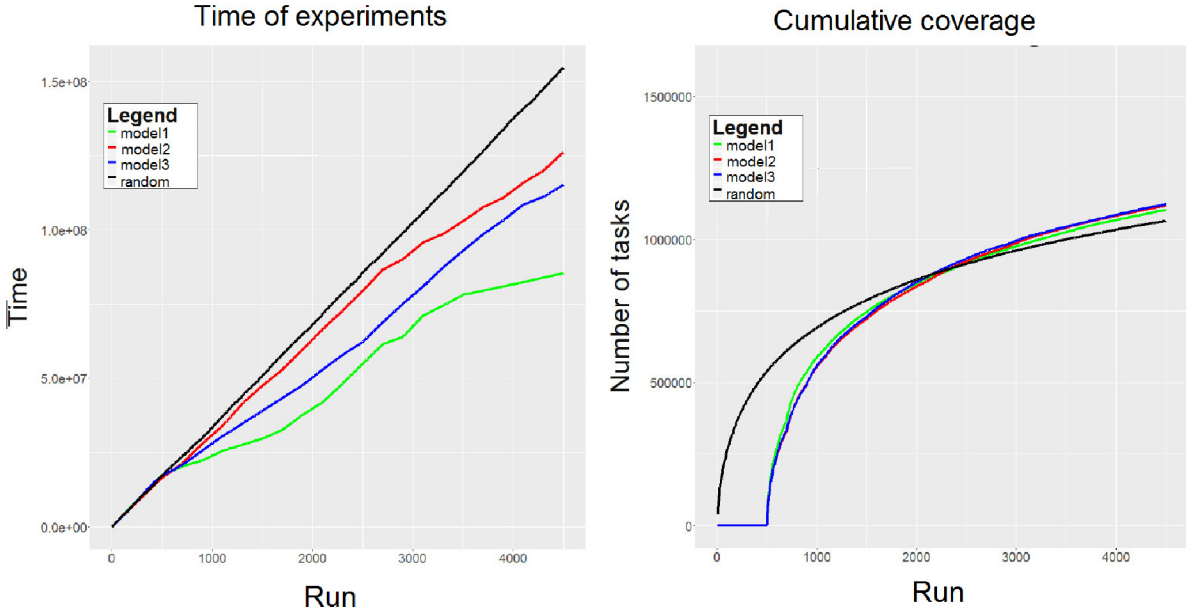



Figure 5.2: Testing time (left) and cumulative coverage (right) for an increasing number of test runs.

Figure 5.2 (right) compares how the obtained cumulative coverage, averaged over all of our case studies, raises when increasing the number of performed test runs under the purely random noise-based approach and under our optimized approaches. All our approaches have better cumulative results despite having an initial penalty because of the use of a number of test runs to find suitable test and noise parameters via cheap metrics. The left part of the figure compares the average time needed by the four approaches over all case studies.

We can see that the optimized approach using only two cheap metrics has the best results (green line).

## 5.7 Combination of Genetic Algorithms and Prediction of Given Metrics

In the following section, we want to use the previous results in the genetic algorithms, whose usage in noise-based testing we discussed previously. Our goal is to increase the performance of the GAs which are commonly very time-consuming. The idea is that we first apply the GA with a fitness function based on the cheap metrics and we run a number of generations. Then we use the results from the first application of the GA as the input generation of another application of the GA, but this time with a fitness function based on the expensive metrics and we run next generations.

We compare this approach with the classic execution of GA, i.e. the case where the GA algorithm executes all generations with only one fitness function based on the given expensive metrics. We assume that the experiments will show acceleration of the optimization process used by the genetic algorithms.

### 5.7.1 Results of Experiments with Predicted Coverage and Genetic Algorithms

For the experiments, we use the Airlines, Crawler, Elevator, HEDC, and Rover case studies only because of their fast executions. We have two types of fitness functions. The first is a fitness function based on cheap metrics:

$$fitness_{cheap} = 1.039 + 0.012 * SYNCHRO + 0.00076 * SHVARPAIRT,$$

the second one is a classic fitness function with the given, more expensive metrics and time:

$$fitness_{classic} = \frac{GoldiLockSC^*}{1443} + \frac{WConcurPairs}{2811899} + \frac{HBPair^*}{120} + \frac{3504127 - time}{3504127}.$$

In the experiments, we used 50 generations of the populations for the classic GA with $fitness_{classic}$. We divided this number of generations into two sub-generations, combining the GAs with the predicted coverage, where the first set of generations uses $fitness_{cheap}$ and the next generations use $fitness_{classic}$. In our case, we tried to use an extreme division: 49 generations were generated with $fitness_{cheap}$ and the last one was executed with $fitness_{classic}$.

Table 5.4 shows the results of the experiments with the coverage of the expensive metrics weighted by the total consumed testing time.

An average improvement of the cumulative metric coverage over time in the new approach is more than 50 % for two of the considered metrics. Only in the case WConcurPairs metric, the result of the coverage decreased a little on average.

To sum up the results, the time needed for the experiments was on average about 15 % worse when using GA combined with the predicted coverage than in the classic setting of GA over all the benchmarks. On the other hand, the sum of the coverage for the individual metrics over all the benchmarks was increased. The improvement is between 3 % and 21 %. An interesting question for the future is how to improve the results by finding the ideal ratio between the number of generations executed under the fitness function with cheap metrics and the number of generations with the fitness function based on the expensive metrics.

Table 5.4: A comparison of GA for coverage metrics over the total testing time.

| | GoldiLockSC* | | WConcurPairs | |
|---|---|---|---|---|
| *CaseStudies* | classic GA | predict GA | classic GA | predict GA |
| Airlines | **0.0094** | 0.0060 | **1.2045** | 0.8229 |
| Crawler | 0.0409 | **0.0511** | **12.1012** | 11.1557 |
| Elevator | 0.0305 | **0.0391** | 9.0522 | **10.1673** |
| HEDC | **0.0209** | 0.01496 | **23.3262** | 15.7001 |
| Rover | 0.2469 | **1.0443** | 24.6319 | **34.3079** |
| Average Impr. | | 1.6234 | | 0.9588 |

| | HBPair* | |
|---|---|---|
| *CaseStudies* | classic GA | predict GA |
| Airlines | **0.0028** | 0.0020 |
| Crawler | 0.0172 | **0.0197** |
| Elevator | 0.0047 | **0.0055** |
| HEDC | **0.0012** | 0.0008 |
| Rover | 0.0239 | **0.0952** |
| Average Impr. | | 1.5412 |

## 5.8 Conclusion and Future Work

We have proposed an approach that uses correlations between cheap and expensive concurrency metrics to optimize the noise-based testing under expensive metrics by finding suitable values of test and noise parameters for such testing through experiments with cheap metrics. Our experiments have shown that such an approach can improve the noise-based testing. In the future, it would be interesting to generalize the idea of finding suitable noise settings maximizing the coverage under an expensive metric via experiments with a cheap one to a context of dealing with other kinds of cheap and expensive analyses some parameters of which may also be correlated.

# Chapter 6

# Conclusion

The goal of this PhD thesis was to propose new approaches to analyze and verify real-life multi-threaded programs, i.e., programs that can be large and that can use many different features, focusing especially on rarely manifesting synchronization errors. It is very difficult to find such errors due to their appearance in very specific interleavings of the threads only.

There exist various ways how to increase the chance of finding such errors during testing. In particular, we used the noise-injection technique for this purpose. This technique can „stress“ running programs so that during their execution, less common thread interleavings are executed. Noise-injection based testing is quite light-weight compared with other approaches, and so it scales well and can cope with many different programs features. However, it comes with some problems too. One of the problems is a large number of combinations how to set up the test and noise parameters for analyzing programs among which it is difficult to find the right ones. This problem is the one that we worked on this thesis.

Previously, genetic algorithms were proposed as a way of finding the best solution of setting the test and noise parameters (instead of choosing them randomly, which is also often used). In particular, the single-objective genetic algorithm (SOGA) was used in the previous work. In this work, we proposed usage of the multi-objective genetic algorithm (MOGA) instead and shown how it can be used in the given domain. We have then shown that MOGA can indeed deliver better results than both the random approach and the single-objective genetic algorithm. One of the major reasons for that is that, in the MOGA case, the individuals do not degenerate during the generation process, i.e., the generation of individuals do not lose diversity. Such a loss of diversity can have a negative impact on the ability of the approach to test different program behaviour because the evolution could get stuck in the local extreme. For the SOGA, it is difficult to combine the different objectives that are typically present in the TNCS problem and whose wrong setting can lead to degeneration. Moreover, we have also proposed a penalization scheme to increase the number of tested uncommon behaviours. Apart from that the experiments showed that MOGA has more stable results than SOGA and random approaches.

Next, for the same goal, we proposed a use of data mining, in particular, the AdaBoost algorithm. Using this data mining method enabled us to find which parameters and their specific values the most affect testing of parallel programs using noise injection for a particular testing goal. On the other hand, it gives us also information about which setting of parameters has not any effect on the testing. We also tried to combine both approaches—AdaBoost and genetic algorithms. In our comparisons of random, AdaBoost,

genetic algorithms, and a combination of the approaches, the best solution was produced by the AdaBoost and its combination with genetic algorithms.

In the further part of the thesis, our work focused on the time needed for finding suitable test and noise settings by experiments with different coverage metrics which are focused on synchronization in concurrent programs. We found that some metrics used for controlling the testing process need a large number of experiments to find right test and noise parameter setting for maximizing coverage under them while other metrics are cheaper. We found some correlation between these expensive and cheap metrics and we proposed a way of how these correlations can be used. In particular, we showed that one can an avoid costly experiments with testing under expensive metrics to find suitable test settings by performing the experiments with cheaper metrics and then using the discovered settings for final testing under the expensive metrics. We used the same principle for the case of testing under multiple metrics at the same time. We realized that the ideal number of cheap metrics which predict a given combination of more expensive metrics is two. The discovered knowledge has been useful also when using genetic algorithms to find the right noise settings.

**Future research directions.** One of the most promising directions of the future research would be an as efficient as possible combination of static and dynamic analyzes. Following this direction which is still in progress, we implemented new heuristics which could be more precise in injecting noise into program execution. In particular, they allow one to choose concrete points in the program or concrete types of points (such as usage of some concrete variables, classes, etc.) where to put noise. Such places could be identified via static analysis as the first step of program verification. The second step would then be dynamic analysis focusing the noise on concrete places, classes, or variables which are identified by the static analysis.

In the process of implementation of the new heuristics, we also tried to replace the IBM ConTest tool by some other technology in the testing process supported by SearchBestie. The reason is that the development of the IBM ConTest tool was stopped some time ago, and the tool is not even maintained any more. For this purpose, we chose RoadRunner which is an open source tool, and it is still being developed.

RoadRunner is a tool which was developed at University of California at Santa Cruz and Williams College as an efficient solution for concurrent program testing. As it was written in [31], the goal of RoadRunner is to provide a robust and flexible framework that substantially reduces the overhead of implementing dynamic analyses. RoadRunner manages the messy, low-level details of dynamic analysis and provides a clean API for communicating an event stream to back-end analysis tools. Each event describes some operation of interest performed by the target program, such as accessing memory, acquiring a lock, forking a new thread, etc. This separation of concerns allows the developer to focus on the essential algorithmic issues of a particular analysis, rather than on orthogonal infrastructure complexities.

The cooperation of the RoadRunner and SearchBestie was described in the bachelor's thesis written by David Kozák [55], where the author of this thesis helped with supervision and follow-up research. Unfortunately, this research is not further developed due to a loss of the collaborating MSc student.

# Index

# Bibliography


[1] Y. Adler, N. Behar, O. Raz, O. Shehory, N. Steindler, S. Ur, and A. Zlotnick. Code Coverage Analysis in Practice for Large Systems. In *Proc. of ICSE'11*, pages 736–745. ACM, 2011.

[2] E. Alba and F. Chicano. Finding Safety Errors with ACO. In *Proc. of GECCO'07*, ACM, 2007.

[3] E. Alba, F. Chicano, M. Ferreira, J. Gomez-Pulido. Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms, In *Proc. of GECCO'08*, ACM, pages 1735–1742, 2008.

[4] E. Alba, F. Chicano. Searching for Liveness Property Violations in Concurrent Systems with ACO, In *Proc. of GECCO'08*, ACM, pages 1727–1734, 2008.

[5] J. Anděl. Statistické metody, Praha: MATFYZPRESS,1st edition, 1993.

[6] R. Avros, V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, S. Ur, T. Vojnar, and Z. Volkovich. Boosted Decision Trees for Behaviour Mining of Concurrent Programs. In *Proc. of MEMICS'14*, pages 15–27. NOVPRESS, 2014.

[7] R. Avros, V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, S. Ur, T. Vojnar, and Z. Volkovich. Boosted Decision Trees for Behaviour Mining of Concurrent Programs. In *Concurrency and Computation: Practice and Experience*. Wiley, 2017.

[8] R. Azizur, T. Mayooran, and G. Mathew. Comparing Linear, Ridge and Lasso Regressions, 10.13140/RG.2.2.30269.77282, 2018.

[9] T. Ball, S. Burckhardt, K. E. Coons, M. Musuvathi, and S. Qadeer. Preemption Sealing for Efficient Concurrency Testing. In *Proc. of TACAS'10*, volume 6015 of LNCS, pages 420–434. Springer-Velrlag, 2010.

[10] Y. Ben-Asher, Y. Eytani, E. Farchi, S. Ur, Shmuel. Noise Makers Need to Know Where to be Silent – Producing Schedules That Find Bugs, In *Proc. of ISOLA'06*, IEEE, pages 458–465, 2006.

[11] S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of PADTAD'05*, LNCS 3875, pages 208–223, 2005.

[12] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM, 2005.

[13] M. Budíková, M. Králová and B. Maroš. Průvodce základními statistickými metodami, Praha: Grada Publishing, 1st edition, 2010.

[14] S.-Ch. Cheung, W. Chen, Y. Liu, and Ch. Xu. CUSTODES: Automatic Spreadsheet Cell Clustering and Smell Detection using Strong and Weak Features, In *Proc. of ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.

[15] F. Chicano, M. Ferreira, E. Alba. Comparing Metaheuristic Algorithms for Error Detection in Java Programs, In *Proc. of SSBSE'11*, volume 6956 of LNCS, Springer-Verlag, pages 82–96, 2011.

[16] H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy Based Testing. In *Proc. of FMCAD '07*, IEEE, 2007.

[17] K. Christensen. Population enetics by Knud Christensen, second edition, 2003. Available in *http://www.ihh.kvl.dk/htm/kc/popgen/genetics/genetik.htm.*

[18] P. Ciancarini, F. Poggi, D. Rossi, and A. Sillitti. Mining Concurrency Bugs, *Embedded Multi-Core Systems for Mixed Criticality Summit*, CPS Week, 2016.

[19] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[20] K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms.* Wiley paperback series. Wiley, 2009.

[21] J.L. Devore. Probability and Statistics for Engineering and the Sciences, Enhanced Review Edition. In *Available 2010 Titles Enhanced Web Assign Series*, Brooks/Cole, Cengage Learning, 2008, ISBN 9780495557449.

[22] R. Dias, C. Ferreira, J. Fiedor, J. Lourenço, A. Smrčka, D.G. Sousa, and T. Vojnar. Verifying Concurrent Programs using Contracts. In *Proc. of ICST'17.* IEEE CS, 2017.

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

[24] 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*, 15(3-5):485–499. Wiley, 2003.

[25] F. Eichinger, V. Pankratius, P. W. L. Große, and K. Böhm. Localizing Defects in Multithreaded Programs by Mining Dynamic Call Graphs. In *Proc. of TAIC PART'10*, volume 6303 of LNCS, pages 56–71. Springer-Velrlag, 2010.

[26] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, pages 245–255. ACM, 2007.

[27] N. Erman, V. Tufvesson, M. Borg, A. Ardö, and P. Runeson. *Navigating Information Overload Caused by Automated Testing – a Clustering Approach in Multi-Branch Development*, ICST'15, IEEE, 2015.

[28] Y. Eytani, Yaniv. Concurrent Java Test Generation as a Search Problem, In *Electronic Notes in Theoretical Computer Science*, volume 144, pages 57–72, Elsevier Science Publishers B. V., 2006.

[29] J. Fiedor, V. Hruba, B. Krena, T. Vojnar. DA-BMC: a Tool Chain Combining Dynamic Analysis and Bounded Model Checking, In *Proc. of RV'11*, LNCS 7186. Springer-Verlag, 2012.

[30] J. Fiedor, V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Advances in Noise-based Testing of Concurrent Software. In Software Testing, Verification and Reliability, 25(3):272–309. Elsevier, 2015.

[31] C. Flanagan, S. N. Freund. The ROADRUNNER Dynamic Analysis Framework for Concurrent Programs. *PASTE'10*, Toronto, Ontario, Canada, 2010. ACM.

[32] Y. Freund. Boosting a weak learning algorithm by majority. In *Information and Computation*, 121(2):256–285,1995.

[33] Y. Freund and R. E. Schapire. *A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting*. Journal of Computer and System Sciences, 55(1): 119–139, Academic Press, Inc., Orlando, FL, USA, August 1997.

[34] Y. Freund and R. E. Schapire. A Short Introduction to Boosting. In *In Proc. of IJCAI'99*, pages 1401–1406. Morgan Kaufmann, 1999.

[35] L. Gao, K. Schulz and M. Bernhardt. Statistical Downscaling of ERA-Interim Forecast Precipitation Data in Complex Terrain Using LASSO algorithm. *Hindawi Publishing Corporation, Adavnces in Meterology*, Article ID 472741, 16 pages, Volume 2014.

[36] C. E. Ginestet. Regularization: Ridge Regression and Lasso. *MA 575 Linear Models, Week 12, Lecture 2*, Boston University.

[37] P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.

[38] A. E. Hassan and T. Xie. Mining Software Engineering Data, In *Proc. of ICSE'10*, pages 503 – 504, ACM New York, NY, USA, 2010.

[39] T. Hastie, R. Tibshirani, and J. Friedman. The Elements of Statistical Learning Data Mining, Inference, and Prediction. Springer, 2009.

[40] S. Hong, J. Ahn, S. Park, M. Kim and M. J. Harrold. Framework for Testing Multi-threaded Java Programs, In *Concurrency and Computation: Practice and Experience*, 15(3-5), John Wiley & Sons, Ltd., 2003.

[41] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of ISSTA'12*. ACM, 2012.

[42] V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, volume 7515 of LNCS, pages 152–167. Springer-Velrlag, 2012.

[43] V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, and T. Vojnar. *Testing Concurrent Programs Using Multi-objective Genetic Algorithms*, FIT-TR-2013-05, Brno, 2013.

[44] V. Hrubá, B. Křena, Z. Letko, H. Pluháčková, and T. Vojnar. Multi-objective Genetic Optimization for Noise-based Testing of Concurrent Software. In *Proc. of SSBSE'14*, volume 8636 of LNCS, pages 107–122. Springer-Verlag, 2014.

[45] G.-H. Hwang, H.-Y. Lin, S.-Y. Lin, Ch.-S. Lin. Statement-Coverage Testing for Concurrent Programs in Reachability Testing, In *Journal of Information Science and Engineering*, Volume 30, number 4, pages 1095-1113, 2014.

[46] G.-H. Hwang, Ch.-S. Lin, T.-S. Lee, Ch. Wu-Lee. A model-free and state-cover testing scheme for semaphore-based and shared-memory concurrent programs, In *Software Testing, Verification and Reliability*, Volume 24, Issue 8, pages 706–737, December 2014.

[47] H. Ishibuchi and Y. Shibata. *A Similarity-based Mating Scheme for Evolutionary Multiobjective Optimization*, In *Proc. of GECCO'03*, LNCS 2723, pages 1065–1076, Springer, 2003.

[48] H. Jain and K. Deb. An Improved Adaptive Approach for Elitist Nondominated Sorting Genetic Algorithm for Many-Objective Optimization. In Evolutionary Multi-Criterion Optimization (EMO), 307–321, 7th International Conference, EMO 2013, Sheffield, UK.

[49] G. James, D. Witten, T. Hastie, R. Tibshirani. An Introduction to Statistical Learning with Applications in R. Springer, 2013.

[50] Y. Jin and J. Branke. Evolutionary Optimization in Uncertain Environments – A Survey. *IEEE Transactions on Evolutionary Computation*, 9(3), 2005.

[51] T. Johnson, N. Barton. Theoretical models of selection and mutation on quantitative traits. In *Biological Sciences*, volume 360, 1459, p. 1411–1425; 2005.

[52] P. Joshi, Ch.-S. Park, K. Sen, M. Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks, In *Proc. of PLDI'09*, pages 110–120, ACM, 2009.

[53] P. Joshi, M. Naik, C.-S. Park, K. Sen. CalFuzzer: An Extensible Active Testing Framework for Concurrent Programs, In *Proc. of CAV'09*, LNVS 5643. Springer-Verlag, 2009.

[54] G. Korl and N. Shavit and P. Felber. *Noninvasive concurrency with Java STM.*

[55] D. Kozák. *Přesné heuristiky pro vkládání šumu v nástroji SearchBestie.* Bachelor's thesis, University of Technology, Faculty of Informatic science, Brno 2017.

[56] P. Kreutzer, G. Dotzler, M. Ring, b. M. Eskofier, and M. Philippsen. Automatic Clustering of Code Changes. In *Proc. of MSR'16*, IEEE/ACM, 2016, Austin, TX, USA.

[57] W. J. Krzanowski and D. J. Hand. *ROC Curves for Continuous Data*, Monographs on Statistics and Applied Probability 111 2009, Chapman & Hall/CRC.

[58] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-the-fly. In *Proc. of PADTAD'07*, ACM, 2007.

[59] 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 RV'09*, LNCS 5779, Springer-Verlag, 2009.

[60] B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*, ACM, 2010.

[61] B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*, volume 7186 of LNCS, pages 177–192. Springer-Velrlag, 2012.

[62] B. Křena, Z. Letko, and T. Vojnar. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*, volume 7119 of LNCS, pages 123–131, Springer-Velrlag, 2012.

[63] B. Křena and T. Vojnar. Automated Formal Analysis and Verification: An Overview In International Journal of General Systems, 42(4):335–365. Taylor and Francis, 2013.

[64] B. Křena, H. Pluháčková, S. Ur, and T. Vojnar. Prediction of Coverage of Expensive Concurrency Metrics Using Cheaper Metrics In *Proc. of EUROCAST'17*, pages 99–108, ISBN 978-3-319-74726-2, Springer-Velrlag, 2018.

[65] J. Kwanghue, D. Amarmend, Y. Geunseok, L. Jung-Won, and L. Byungjeong. Bug Severity Prediction by Classifying Normal Bugs with Text and Meta-Field Information. *Advanced Science and Technology Letters*, 129, Mechanical Engineering, 2016.

[66] B. Lantz. Machine Learning with R, *Birmingham Packt Publishing*, 2013.

[67] J. Laski and W. Stanley. Software Verification and Analysis, Springer-Verlag , 2009.

[68] Z. Letko. Analysis and Testing of Concurrent Programs, Faculty of Information Technology BUT, Brno, CZ, 2012.

[69] S. Leue, A. Stefanescu, W. Wei. A Livelock Freedom Analysis for Infinite State Asynchronous Reactive Systems, In *CONCUR 2006 – Concurrency Theory*, series = Lecture Notes in Computer Science, Department of Computer and Information Science, University of Konstanz, D-78457 Konstanz Germany Germany, Springer Berlin / Heidelberg, volume 4137, pages 79–94, 2006.

[70] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Can. AntMiner: Mining More Bugs by Reducing Noise Interference. In *Proc. of ICSE'16*,IEEE/AMC, 2016, Austin, TX, USA.

[71] Ch.-S. Lin and G.-H. Hwang. State-cover Testing for Nondeterministic Concurrent Programs with an Infinite number of Synchronization Sequences, In *SCIENCE OF COMPUTER PROGRAMMING*, Volume 78, Issue 9, 1 September 2013, Pages 1294–1323.

[72] S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*. ACM, 2006.

[73] S. Lu, W. Jiang, Y. Zhou. A study of interleaving coverage criteria, In *ESEC-FSE companion'07*, ACM, pages 533??536, 2007.

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

[75] S. Luke. *Essentials of Metaheuristics*, first, 2011.
Available for free at http://cs.gmu.edu/∼sean/book/metaheuristics/

[76] Q. Luo, D. Poshyvanyk, and M. Grechanik. Mining Performance Regression Inducing Code Changes in Evolving Software. In *Proc. of MSR'16*, IEEE/ACM, 2016, Austin, TX, USA.

[77] O. Morozova, O. Levina, A. Uusküla, and R. Heimer Comparison of subset selection methods in linear regression in the context of health-related quality of life and substance abuse in Russia, BMC Medical Research Methodology (2015) 15:71.

[78] M. Musuvathi, S. Qadeer, T. Ball. CHESS: a Systematic Testing Tool for Concurrent Software, Microsoft Research, MSR-TR-2007-149, 2007.

[79] F. Nielson, H. R. Nielson and C. Hankin. Principle of Program Analysis, Springer-Verlag , 2005.

[80] R. V. van Nieuwpoort *Efficient Java-Centric Grid Computing*, 2003, Rob van Nieuwpoort. Available for free at https://books.google.cz/books?id=fRmZ6aC4eDwC

[81] Y. Nir-Buchbinder, R. Tzoref, S. Ur. *Deadlocks: From Exhibiting to Healing*, In Third Workshop on Runtime Verification (RV03), volume 89(2) of Electronic Notes in Theoretical Computer Science, 2004.

[82] S. Park, S. Lu, Y. Zhou. CTrigger: Exposing Atomicity Violation Bugs from Their Hiding Places, In *Proc. of ASPLOS'09*, ACM Press, 2009.

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

[84] Ch. von Praun and T. R. Gross. *Object Race Detection*, SIGPLAN Not., 2001, volume 36, number 11, pages 70 –82, New York, NY, USA, ACM.

[85] Ch. von Praun and T. R. Gross. *Static detection of atomicity violations in object oriented programs*, Runtime Verification, 2008, pages 104–118, Springer Berlin Heidelberg.

[86] E. Rubinić, G. Mauša, and T. Galinac Grbac. *Software Defect Classification with a Variant of NSGA-II and Simple Voting Strategies*, In *Proc. of SSBSE'15*, LNCS 9275, pp.347–353,2015.

[87] P. Saint-Hilaire, Ch. von Praun, E. Stolte, G. Alonso, A. O. Benz and T. Gross. *The RHESSI Experimental Data Center*, Solar Physics 210: 143–164, 2002.

[88] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson. Eraser: a Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*. ACM, 1997.

[89] M. I. Schwartzbach. Lecture Notes on Static Analysis, BRICS, Department of Computer Science, University of Aarhus, Denmark, 2006.

[90] K. Sen. Race Directed Random Testing of Concurrent Programs, In *Proc. of PLDI'08*, ACM, pages 11–21, 2008.

[91] E. Sherman, M. B. Dwyer, S. Elbaum. Saturation-based Testing of Concurrent Programs,, In *Proc. of ESEC/FSE'09*, ACM, pages 53–62, 2009.

[92] L. A. Smith, J. M. Bull, J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Proc. of Supercomputing'01*, ACM, 2001.

[93] J. Staunton and J. A. Clark. Searching for Safety Violations Using Estimation of Distribution Algorithms. In *Proc. of ICSTW'10*, IEEE, 2010.

[94] S. D. Stoller. *Testing Concurrent Java Programs using Randomized Scheduling*, Proc. Second Workshop on Runtime Verification (RV), 2002, series: Electronic Notes in Theoretical Computer Science, volume 70(4), Elsevier.

[95] Ch. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated Parameter Optimization of Classification Techniques for Defect Prediction Models, In *Proc. ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.

[96] G. Tassey. The Economic Impacts of Inadequate Infrastructure for Software Testing, *National Institute of Standards and Technology, RTI Project*, volume 7007, Citeseer, 2002.

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

[98] R. Tibshirani. Regression Shrinkage and Selection via the LASSO. In *Journal of the Royal Statistical Society, Series B*, 58(1):267–288, 1996.

[99] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*, pages 27–38. ACM, 2007. ACM.

[100] K. Wang, and Z. Chen. Stepwise Regression and All Possible Subsets Regression in Education, EIJEAS 2016 Volume: 2 Issue: Special Issue, 60-81, Ohio, USA Electronic International Journal of Education, Arts, and Science

[101] S. Wang, T. Liu, and L. Tan. Automatically Learning Semantic Features for Defect Prediction, In *Proc. of ICSE'16*, IEEE/ACM, 2016, Austin, TX, USA.

[102] D. White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, 13, 2012.

[103] I. H. Witten, E. Frank, and M. A. Hall. *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 3rd edition, 2011.

[104] J. Wu, Y. Tang, G. Hu, H. Cui, and J. Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012.

[105] Y. Xu, P. Liang, and M. A. Babar. *Introducing Learning Mechanism for Class Responsibility Assignment Problem*, In *Proc. of SSBSE'15*, LNCS 9275, pp. 311–317, 2015.

[106] S. Yoo, M. Harman, P. Tonella, and A. Susi. Clustering Test Cases to Achieve Effective and Scalable Prioritisation Incorporating Expert Knowledge. In *Proc. of ISSTA'09*, pages 201–212. ACM, 2009.

[107] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam. Maple: a Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*. ACM, 2012.

[108] E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zurich, 1999.

[109] E. Zitzler, L. Thiele. Multiobjective Evolutionary Algorithms: a Comparative Case Study and the Strength Pareto Approach, 1999.

[110] E. Zitzler, M. Laumanns, L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm, 2001.