BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Information Technology

Department of Intelligent Systems

Mgr. Bc. Hana Pluháčková

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

Využití genetických algoritmů a dolování z dat v testování paralelních programů s využitím šumu

Extended Abstract of a PhD Thesis

Supervisor: Prof. Ing. Tomáš Vojnar, Ph.D.

Key Words

Testing, concurrent programs, data mining, genetic algorihtms, AdaBoost, lasso algorithm, noise injection.

Klíčová slova

Testování, paralelní programy, dolování z dat, genetické algoritmy, AdaBoost, lasso algoritmus, vkládání šumu.

Contents

1 Introduction **5**
1.1 Analysis and Verification of Programs 5
1.2 Verification of Concurrent Programs 7
1.3 Goals of the Thesis 8

2 Preliminaires **9**
2.1 Noise Injection 9
2.1.1 Noise Seeding Heuristics 10
2.1.2 Noise Placement Heuristics 11
2.2 Methods Used in Thesis 12

3 An application of Genetic Algorithms in Noise-based Testing **15**
3.1 Experimental evaluation 16

4 Using Data Mining in Testing of Concurrent Programs **21**
4.1 Analysis of Knowledge Hidden in Obtained Classifiers . 22
4.2 Fully-Automated Noise-based Testing with AdaBoost 25

5 Prediction Coverage of Expensive Metrics from Cheaper Ones **29**
5.1 Using Correlations of Metrics to Optimize Noise-based Testing 31

6 Conclusion **38**

Bibliography **42**

Curriculum Vitae **45**

Abstract **46**

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 [Tas02].

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), *com-*

pleteness (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) [Sch06]. 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 Programs

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 [ECP99], 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*. The noise injection approach [SHH03] 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 [SHH03, KLV12, RDV17] 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 2.1.

1.3 Goals of the 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.

2 Preliminaires

2.1 Noise Injection

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 1, we illustrate two of the possible effects that noise injection can have. Figure 1(a) illustrates a scenario in which 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 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.

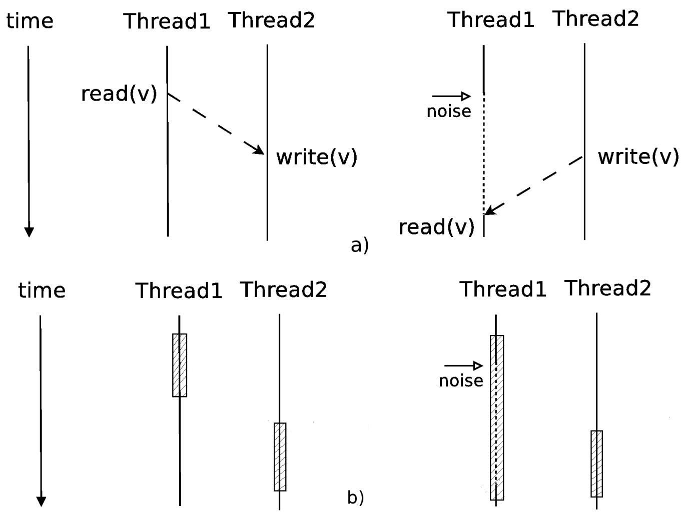

Figure 1: Two examples of the effect of noise injection: (a) reordering of the common order of two events in a concurrent program execution and (b) prolongation of the time spent by a thread in a critical section, leading to an overlapped execution of two critical sections.

2.1.1 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 parameterised 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.

2.1.2 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 Methods Used in Thesis

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.

Data mining methods are next approaches used in this work. 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 [Lan13]. Only two of them, namely classification and regression, are introduced, as those have been used in the methods proposed in this work.

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 Neighbour.

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 Section 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.

The last approaches used in this thesis are *genetic and evolutionary algorithms*. 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 [Zit99]. 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) [Tal09], 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.

3 An application of Genetic Algorithms in Noise-based Testing

Testing of multi-threaded programs is a demanding work due to the many possible thread interleavings one should examine. The noise injection technique helps to increase the number of thread interleavings examined during repeated test executions provided that a suitable setting of noise injection heuristics is used. The problem of finding such a setting, i.e., the so called test and noise configuration search problem (TNCS problem), is not easy to solve according to previous work [DKLV12]. In this section, we show how to apply a multi-objective genetic algorithm (MOGA) to the TNCS problem. In particular, we focus on generation of TNCS solutions that are suitable for regression testing where tests are executed repeatedly. Consequently, we are searching for TNCS candidate solutions that cover a high number of distinct interleavings (especially those which are rare) and provide stable results in the same time. To achieve this goal, we study suitable metrics and ways how to suppress effects of non-deterministic thread scheduling on the proposed MOGA-based approach. We also discuss a choice of a MOGA and its parameters suitable for our setting. Finally, we show on a set of benchmark programs that our approach provides better results when compared to the commonly used random approach as well as to the sooner proposed use of a single-objective genetic approach (SOGA).

3.1 Experimental evaluation

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 [DKLV12] 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.

In the experiments, we used the following settings of the MOGA— 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*.

For SOGA-based approach, we use the following parameters taken from [DKLV12]: size of population 20, number of generations 50, two different selection operators (tournament among four individuals and fitness proportional[1]), 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[2]:

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

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 [DKLV12], 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 2 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 2 clearly shows that our MOGA-based approach does not suffer from the degeneration problem unlike the SOGA-based approach.

[1]Experiments presented in [DKLV12] 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.

[2]In the experiments performed in [DKLV12], the fitness function was sensitive on weight. Therefore, we removed the weight from our new fitness function for SOGA.

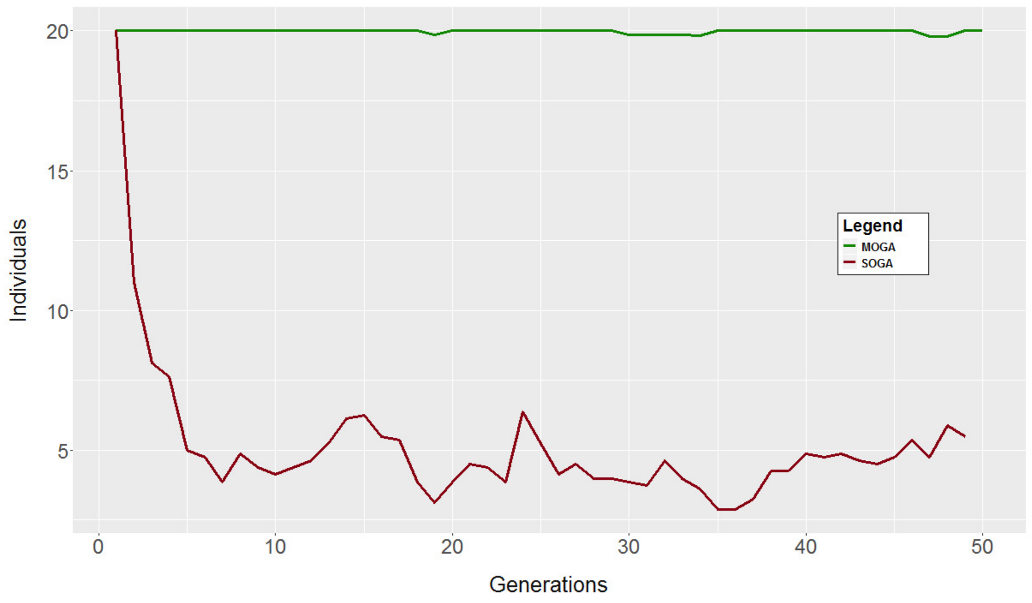

Figure 2: Degeneration of MOGA-based and SOGA-based search processes.

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 [DKLV12]) is that even a degenerated population can provide a high coverage if the repeatedly generated candidate solutions suffer from low stability, which allows them to test different behaviours in different executions.

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 1. 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 1 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 1: 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

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 2 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.

Table 2: Efficiency of the 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	

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 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

Table 3: 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

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.

4 Using Data Mining in Testing of Concurrent Programs

In this section, we propose a novel application of data mining allowing one to exploit information present in data obtained from a sample of test runs of a concurrent program to optimize the process of noise-based testing of the given program. To be more precise, our method employs a data mining method based on *classification* by means of *decision trees* and the *AdaBoost* algorithm. The approach is, in particular, intended to find out which parameters of the available tests and which parameters of the noise injection system are the most influential and which of their values (or ranges of values) are the most promising for a particular testing goal for the given program.

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 (which are, of course, to be validated in the future on more case studies). (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.

4.1 Analysis of Knowledge Hidden in Obtained Classifiers

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: 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), 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 [KLV12] 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 5. 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

Table 5: 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

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 [KLV12, DKL$^+$14, DKLV12] and that the *haltOneThread* noise injection heuristics (x_4) provides in some cases a dramatic increase in the probability of spotting an error [KLV12]. 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.

4.2 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. 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 (Random), generation based on genetic algorithms (SOGA and MOGA), random generation filtered through the classic AdaBoost approach, random generation restricted to the AdaBoost-recognised most influential values of parameters (AdaBoost2), and combination of genetic algorithms and AdaBoost2 approach (SOGA2 and MOGA2).

Table 6: 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.

	Random	*SOGA*	*MOGA*	*AdaBoost*
CaseStudies	error/ %	error / %	error/ %	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

	AdaBoost2	*SOGA2*	*MOGA2*
CaseStudies	error/ %	error/ %	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**

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).

Table 6 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.

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. 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 random approach, we executed 1000 test runs with randomly generated test and noise configurations. For the other ap-

Table 7: A comparison of average cumulative numbers of rare tasks over the time needed to cover them.

	Rand.	SOGA	MOGA	AdaBoost
CaseStudies	rareTasks/ %	rareTasks/ %	rareTasks/ %	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

	AdaBoost2	SOGA2	MOGA2
CaseStudies	rareTasks/ %	rareTasks/ %	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

proaches, 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 7, 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.

5 Prediction Coverage of Expensive Metrics from Cheaper Ones

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 [FDK$^+$15]. 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 [DKLV12, DKL+14, ADK+14]. 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.

5.1 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 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 [DKLV12, DKL^{+}14], or data mining on the test results can be used [ADK^{+}14], all the time evaluating the performed experiments via the chosen cheap metrics, or, more precisely, through the predictive model built. In the simplest case, one can perform just a number of random experiments with different test and noise settings and choose the settings that performed the best in these experiments wrt the predictive model. 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 5 test runs with each of them for each of our case studies while collecting coverage under the selected cheap metrics (leading to 500 executions for each case study). We cumulated results within the 5 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 coverage under the expensive metrics for each of the case studies. Hence, both of the approaches were given the same number of test runs.

Table 8: 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 8, 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. From the table, we can see 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 %.

Finally, Figure 4 (left) compares how the obtained cumulative coverage, averaged over all of our case studies, increases when increasing the number of performed test runs under the purely random noise-based approach and under our optimized approach. Our approach wins despite it has some initial penalty due to using some number of

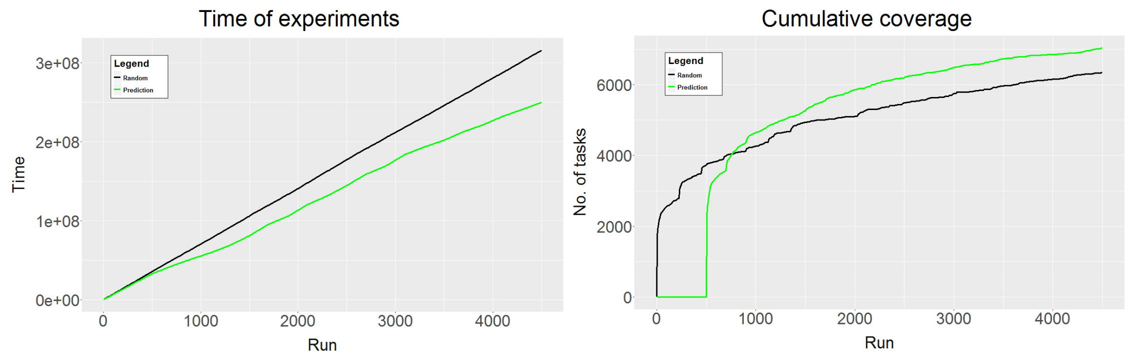

Figure 3: Cumulative coverage (left) and testing time (right) for an increasing number of test runs.

test runs to find suitable test and noise parameters via cheap metrics. The right part of the figure then compares the average time needed by the two approaches over all the case studies. Again, the optimized approach is winning.

Discovering Ideal Number of Cheap Metrics to Increase Performance. In the next experiments, we want to predict some given three different metrics, not only expensive, by cheap metrics. In the genetic algorithm, there is often to use some given fitness function for generating better results. There are three experiments where we try to predict given fitness function by two, three and four cheap metrics and we compare the results. Such experiments show us ideal number of cheap metrics to predict some other metrics.

As in the case of three expensive metrics, we randomly generated 100 test and noise configurations and executed 5 test runs with each of them for each of our case studies while collecting coverage under the selected cheap metrics (leading to 500 executions for each case study). We cumulated results within the 5 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 coverage under the expensive metrics for each of the case studies. Hence, both of the approaches were given the same number of test runs.

In Table 9, we compare the random approach with three prediction-based approaches. From the previous experiment with three expensive metrics, we know that the prediction optimization works relatively good. Now, we want to find how many cheap metrics must be used for prediction to have the best results. From the table, we can see that the results between model1, model2 and model3 are not so much 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.

Finally, Figure 4 (right) compares how the obtained cumulative coverage, averaged over all of our case studies, increases when increasing the number of performed test runs under the purely random noise-based approach and under our optimized approach. Our approach wins despite it has some initial penalty due to using some number of test runs to find suitable test and noise parameters via cheap metrics. The right part of the figure then compares the average time needed by the two approaches over all the case studies. Again, the optimized approach is winning.

Combination of Prediction Given Metrics and Genetic Algorithms. In the following section, we want to use previous results in

Table 9: A comparison of random and three prediction-optimized 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

the genetic algorithms where the goal is increase the performance of GA which are commonly very time consuming. Idea is that firstly, we

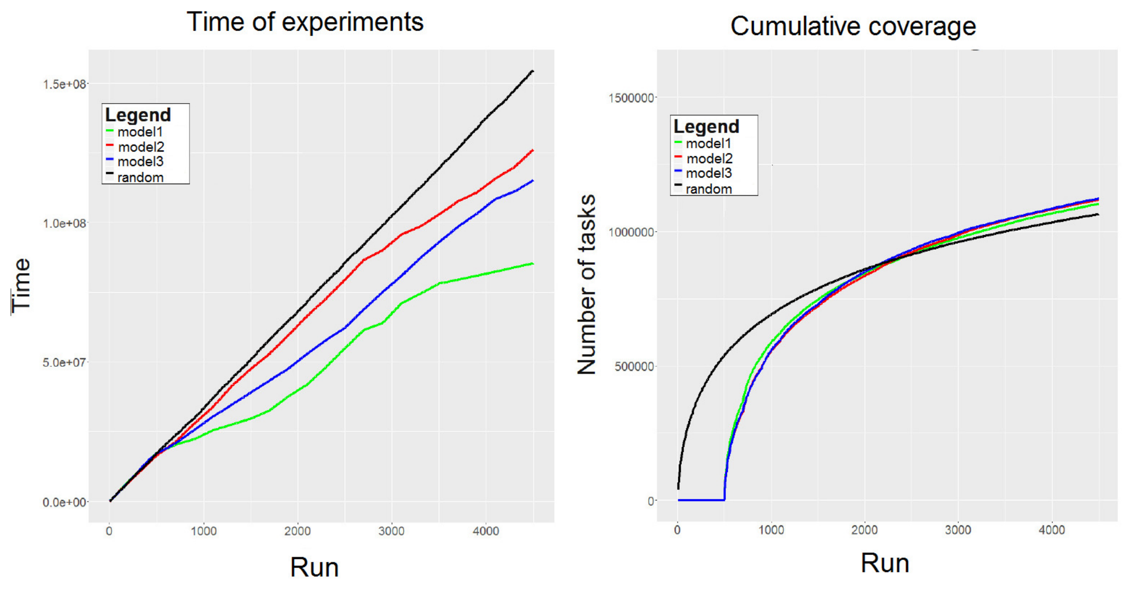

Figure 4: Testing time (left) and cumulative coverage (right) for an increasing number of test runs.

apply GA with a fitness function based on the cheap metrics and we run some number of generations. Then we use the results from the first application of GA as the input generation of another application of GA but this time with a fitness function based on expensive metrics and we run next generations.

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

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

Table 10 shows results of the experiments with the coverage of the expensive metrics weighted by the total consumed testing time.

Table 10: A comparison of the types 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

Average improvement of the cumulative metric coverage over the time in the new approach case is in the two metrics better than 50%. Only in WConcurPairs metric case, the result of coverage decreased a little in average.

To sum up the results, the time needed for the experiments was in average about 15% worse in the prediction methods than in the classic using of GA over the all benchmarks. On the other hand, sum of coverage for individual metrics over the all benchmarks was increased.

The improvement is between 3% and 21%. Some next way how to increase the results is to find the ideal ratio between number of generations executed under the fitness function with cheap metrics and number of generations with the fitness function based on the expensive metrics.

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 loose diversity. 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 [CF10], 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 [Koz17], 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.

Publication Related to this Thesis. The results presented in this thesis were originally published in the following papers. The research focused on MOGA was published as a technical report [DKL+13] in cooperation with ORT Braude College in Karmiel, Israel within the Kontakt II project. The second publication is focusing on the comparison of two genetic approaches MOGA and SOGA and it was presented as a conference paper [DKL+14] at the SSBSE'14 conference. The AdaBoost approach was presented as conference a paper [ADK+14] 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 a journal paper [ADK+17] in the Concurrency and Computation: Practice and Experience journal. 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. The main idea 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 setting suitable for a more expensive metric. Alternatively testing under several cheaper metrics can be used for this purpose too. The main idea about identification of dependencies between parameter settings suitable for testing under metrics of different costs and then use testing under a cheaper metric to find setting suitable for more expensive metrics was presented at the EUROCAST'17 conference and published as a conference paper [KPV18].

Bibliography

[ADK$^+$14] Renata Avros, Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, Shmuel Ur, Tomáš Vojnar, and Zeev Volkovich. Boosted decision trees for behaviour mining of concurrent programs. In *Proceedings of MEMICS'14*, pages 15–27. NOVPRESS s.r.o., 2014.

[ADK$^+$17] Renata Avros, Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, Shmuel Ur, Tomáš Vojnar, and Zeev Volkovich. Boosted decision trees for behaviour mining of concurrent programs. *Concurrency and Computation: Practice and Experience*, 2017.

[CF10] S. N. Freund. C. Flanagan. The roadrunner dynamic analysis framework for concurrent programs. In PASTE'10. Toronto, Ontario, Canada, ACM, 2010.

[DKL$^+$13] Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš Vojnar. Testing concurrent programs using multi-objective genetic algorithms, 2013.

[DKL$^+$14] Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš Vojnar. Multi-objective genetic optimization for noise-based testing of concurrent software. In *SSBSE'14*, LNCS 8636, pages 107–122. Springer Verlag, 2014.

[DKLV12] Vendula Dudka, Bohuslav Křena, Zdeněk Letko, and Shmuel Ur Tomáš Vojnar. Testing of concurrent programs using genetic algorithms. In *SSBSE'12*, LNCS 7515, pages 152–167. Springer Verlag, 2012.

[ECP99] O. Grumberg E. Clarke and D. Peled. Model checking. 1999.

[FDK+15] Jan Fiedor, Vendula Dudka, Bohuslav Křena, Zdeněk Letko, Shmuel Ur, and T. Vojnar. Advances in noise-based testing of concurrent software. In Software Testing, Verification and Reliability, pages 272–309. Elsevier, 2015.

[KLV12] Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. Influence of noise injection heuristics on concurrency coverage. LNCS 7119, pages 123–131. Springer Verlag, 2012. In *Proc. of MEMICS'11*,.

[Koz17] D. Kozák. Přesné heuristiky pro vkládání šumu v nástroji searchbestie. University of Technology, Faculty of Informatic science, Brno, 2017. Bachelor's thesis.

[KPV18] Bohuslav Křena, Hana Pluháčková, and Shmuel Ur Tomáš Vojnar. Prediction of coverage of expensive concurrency metrics using cheaper metrics. pages 99–108. Springer Verlag, 2018. In *Proc. of EUROCAST'17*,.

[Lan13] B. Lantz. Machine learning with r. Birmingham Packt Publishing, 2013. In *Proc. of EUROCAST'17*,.

[RDV17] J. Fiedor J. Lourenço A. Smrčka D.G. Sousa R. Dias, C. Ferreira and T. Vojnar. Verifying concurrent programs using contracts. In Proc. of ICST'17. IEEE CS, 2017.

[Sch06] M. I. Schwartzbach. Lecture notes on static analysis. BRICS, Department of Computer Science, University of Aarhus, Denmark,, 2006.

[SHH03] S. Park M. Kim S. Hong, J. Ahn and M. J. Harrold. Framework for testing multi-threaded java programs. pages 15(3–5). John Wiley & Sons, Ltd., 2003.

[Tal09] E.-G. Talbi. Metaheuristics: From design to implementation. Wiley Publishing, 2009.

[Tas02] G. Tassey. The economic impacts of inadequate infrastructure for software testing. 2002.

[Zit99] E. Zitzler. Evolutionary algorithms for multiobjective optimization: Methods and applications. 1999. PhD thesis.

Curriculum Vitae

Personal Data

Name:	Hana Pluháčková
Born:	September 27, 1988, Brno, Czech Republic
E-mail:	ipluhackova@fit.vutbr.cz
Telephone:	+420 776 010 677

Education

2013	–	now	PhD. (ongoing)—Faculty of Information Technology, Brno University of Technology. Research theme: *Application of genetic algorithms and data mining in noise-based testing of concurrent software.*
2011	–	2013	Master's degree (Mgr.)—Faculty of Science, Brno Masaryk University. Master thesis: *Non-parametric estimates of ROC curves.*
2008	–	2012	Bachelor's degree (Bc.)—Faculty of Science, Brno Masaryk University. Bachelor thesis: *Free groups.*
2008	–	2011	Bachelor's degree (Bc.)—Faculty of Arts, Brno Masaryk University. Bachelor thesis: *F. G. I. Eckstein and paintings above the stairs of Landhouse in Brno.*
2000	–	2008	Secondary education—Gymnázium, Brno, Vídeňská 47.

Experience

2016	–	now	Security data analyst in Mavenir, s.r.o, Brno, CZ.

Language skills

Czech, English, French.

Abstract

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 vkládání 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.