Multi-objective Genetic Optimization for Noise-based Testing of Concurrent Software

Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Hana Pluháčková, and Tomáš Vojnar

IT4Innovations Centre of Excellence, Brno University of Technology, Czech Republic
{ihruba, krena, iletko, ipluhackova, vojnar}@fit.vutbr.cz

Abstract. 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. In this paper, 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 cover a high number of distinct interleavings (especially those which are rare) and provide stable results at 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 concrete 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.

1 Introduction

Multi-threaded software design has become widespread with the arrival of multi-core processors into common computers. Multi-threaded programming is, however, significantly more demanding. Concurrency-related errors such as data races [9], atomicity violations [21], and deadlocks [3], are easy to cause but very difficult to discover due to the many possible thread interleavings to be considered [22, 8]. This situation stimulates research efforts towards advanced methods of testing, analysis, and formal verification of concurrent software.

Precise static methods of verification, such as model checking [6], do not scale well and their use is rather expensive for complex software. Therefore, lightweight static analyses [2], dynamic analyses [9], and especially testing [26] are still very popular in the field. A major problem for testing of concurrent programs is the non-deterministic nature of multi-threaded computation. It has been shown [22, 8] that even repeated execution of multi-threaded tests, when done naïvely, does often miss many possible behaviors of the program induced by different thread interleavings. This problem is targeted by the *noise injection* technique [8] which disturbs thread scheduling and thus increases chances to examine more possible thread interleavings. This approach does significantly improve the testing process provided that a suitable setting of noise injection heuristics

is used. The problem of finding such a setting (together with choosing the right tests and their parameters), i.e., the so-called *test and noise configuration search problem* (TNCS problem), is, however, not easy to solve [13].

In this paper, we propose an application of a *multi-objective genetic algorithm* (MOGA) to solve the TNCS problem such that the solutions provide high *efficiency* and *stability* during repeated executions. By *efficiency*, we mean an ability to examine as much existing and important program behavior with as low time and resource requirements as possible. On the other hand, *stability* stands for an ability of a test setting to provide such efficient results in as many repeated test executions as possible despite the scheduling non-determinism. Such requirements on the tests and testing environment (and hence noise generation) can be useful, for instance, in the context of *regression testing* [26], which checks whether a previously working functionality works in a new version of the system under test too and which is executed regularly, e.g., every night.

Our proposal of a MOGA-based approach for testing of concurrent programs aims both at high efficiency as well as stability, i.e., we search for such tests, test parameters, noise heuristics, and their parameters that examine a lot of concurrency behavior in minimal time and that provide such good results constantly when re-executed. With that aim, we propose a multi-objective fitness function that embeds objectives of different kinds (testing time, coverage related to finding common concurrency errors like data races and deadlocks, as well as coverage of general concurrency behavior). Moreover, the objectives also embed means for minimizing the influence of scheduling non-determinism and means emphasizing a desire to search for less common behaviors (which are more likely to contain not yet discovered errors). Further, we discuss a choice of one particular MOGA from among of several known ones—in particular, the *Non-Dominated Sorting Genetic Algorithm II* (NSGA-II) and two versions of the *Strength Pareto Evolutionary Algorithm* (SPEA and SPEA2)—as well as a choice of a configuration of its parameters suitable for our setting. Finally, we show on a number of experiments that our solution provides better results than the commonly used random setting of noise injection as well as the approach of solving the TNCS problem via a single-objective genetic algorithm (SOGA) presented in [13].

2 Related Work

This section provides a brief overview of existing approaches for testing and dynamic analysis of multi-threaded programs as well as of applications of meta-heuristics to the problems of testing and analysis of multi-threaded programs.

Testing Multi-Threaded Programs. Simple *stress testing* based on executing a large number of threads and/or executing the same test in the same testing environment many times has been shown ineffective [23, 22, 8]. To effectively test concurrent programs, some way of influencing the scheduling is needed. The *noise injection* technique [8] influences thread interleavings by inserting delays, called *noise*, into the execution of selected threads. Many different noise heuristics can be used for this purpose [20]. 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 [20]. 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 [17]. A proper selection of the noise heuristics and their parameters is not easy, and it is often done by random. In this paper, we strive to improve this practice by applying multi-objective genetic optimization.

An alternative to noise-based testing is *deterministic testing* [12, 22, 30] which can be seen as execution-based model checking. This approach is based on deterministic control over the scheduling of threads, and it can guarantee a higher coverage of different interleavings than noise-based testing. On the other hand, its overhead may be significantly higher due to a need of computing, storing, and enforcing the considered thread interleavings. Since the number of possible interleavings is usually huge, the approach is often applied on abstract and/or considerably bounded models of the SUT. It is therefore suitable mainly for unit testing.

Both of the above mentioned approaches can be improved by combining them with *dynamic analysis* [9, 21, 3] which collects various pieces of information along the executed path and tries to detect errors based on their characteristic symptoms even if the errors do themselves not occur in the execution. Many problem-specific dynamic analyses have been proposed for detecting special classes of errors, such as data races [9], atomicity violations [21], or deadlocks [3].

Meta-heuristics in Testing of Concurrent Programs. A majority of existing works in the area of search-based testing of concurrent programs focuses on applying various meta-heuristic techniques to control state space exploration within the *guided (static) model checking* approach [11, 27, 1]. The basic idea of this approach is to explore areas of the state space that are more likely to contain concurrency errors first. The fitness functions used in these approaches are based on detection of error states [27], distance to error manifestation [11], or formula-based heuristics [1] 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.

Applications of meta-heuristics in deterministic testing of multi-threaded programs are studied in [5, 28]. In [5], a cross entropy heuristic is used to navigate deterministic testing. In [28], an application of a genetic algorithm to the problem of unit test generation is presented. The technique produces a set of unit tests for a chosen Java class. Each unit test consists of a prefix initialising the class (usually, a constructor call), a set of method sequences (one sequence per thread), and a schedule that is enforced by a deterministic scheduler.

In [13], which is the closest to our work, a SOGA-based approach to the TNCS problem is proposed and experimentally shown to provide significantly better results than random noise injection. On the other hand, the work also shows that combining the different relevant objectives into a scalar fitness function by assigning them some fixed weights is problematic. For instance, some tests were sometimes highly rated due to their very quick execution despite they provided a very poor coverage of the SUT behavior. Further, it was discovered that in some cases, the genetic approach suffered from degradation, i.e., a quick loss of diversity in population. Such a loss of diversity

can unfortunately have a negative impact on the ability of the approach to test different program behaviors. Finally, it turned out that candidate solutions which were highly rated during one evaluation did not provide such good results when reevaluated again. In this paper, we try to solve all of the above problems by using a MOGA-based approach enhanced by techniques intended to increase the stability of the approach as well as to stress rare behaviors.

3 Background

In this section, we briefly introduce multi-objective genetic algorithms, the TNCS problem, and the considered noise injection heuristics. Moreover, we provide an overview of our infrastructure and test cases used for an evaluation of our approach.

Multi-objective Genetic Algorithms (MOGA). The genetic algorithm is a biology-inspired population-based optimization algorithm [31, 7]. The algorithm works in iterations. In each iteration, a set of candidate solutions (i.e., individuals forming a population) is evaluated through a fitness function based on some chosen objective(s). The obtained fitness is then used by a selection operator to choose promising candidates for further breeding. The breeding process employs crossover and mutation operators to modify the selected individuals to meet the exploration and/or exploitation goals of the search process.

In single-objective optimization, the set of candidate solutions needs to be totally ordered according to the values of the fitness function. The traditional approach to solve a multi-objective problem by single-objective optimization is to bundle all objectives into a single scalar fitness function using a *weighted sum of objectives*. The efficiency of this approach heavily depends on the selected weights which are sometimes not easy to determine.

On the other hand, multi-objective optimization treats objectives separately and compares candidate solutions using the *Pareto dominance* relation. A MOGA searches for non-dominated individuals called *Pareto-optimal* solutions. There usually exists a set of such individuals which form the *Pareto-optimal front*. Solutions on the Pareto-optimal front are either best in one or more objectives or represent the best available trade-off among considered objectives. There exist several algorithms for multi-objective optimization that use different evaluation of individuals, but all of them exploit the non-dominated sorting. In this paper, we consider the *Non-Dominated Sorting Genetic Algorithm II* (NSGA-II) [7] and two versions of the *Strength Pareto Evolutionary Algorithm* (SPEA and SPEA2) [31].

The Test and Noise Configuration Search Problem. The *test and noise configuration search problem* (the TNCS problem) is formulated in [13] 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 certain test objectives. 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*. The TNCS problem can now be seen as searching for a test configuration from TC according to given objectives.

Considered Noise Injection Heuristics. We consider 6 basic and 2 advanced noise seeding techniques that are all commonly used in noise-based testing [20]. The basic techniques cannot be combined, but any basic technique can be combined with one or both advanced techniques. The basic heuristics are: *yield*, *sleep*, *wait*, *busyWait*, *synchYield*, and *mixed*. The *yield* and *sleep* techniques inject calls of the `yield()` and `sleep()` functions. The *wait* technique injects a call of `wait()`. The concerned threads must first obtain a special shared monitor, then call `wait()`, and finally release the monitor. The *synchYield* technique combines the yield technique with obtaining the monitor as in the *wait* approach. The *busyWait* technique inserts code that just loops for some time. The *mixed* technique randomly chooses one of the five other techniques at each noise injection location. Next, the first of the considered advanced techniques, the *haltOneThread* technique, occasionally stops one thread until any other thread cannot run. Finally, the *timeoutTamper* heuristics randomly reduces the time-outs used in the program under test in calls of `sleep()` (to ensure that they are not used for synchronisation). These heuristics can be used with different *strength* in the range of 0–100. The meaning of the strength differs for different heuristics—it means, e.g., how many times the yield operation should be called when injected at a certain location, for how long a thread should wait, etc.

Further, we consider 3 noise placement heuristics: the *random* heuristics which picks program locations randomly, the *sharedVar* heuristics which focuses on accesses to shared variables, and the *coverage-based* heuristics [20] which focuses on accesses near a previously detected context switch. The *sharedVar* heuristics has two parameters with 5 valid combinations of its values. The *coverage-based* heuristics is controlled by 2 parameters with 3 valid combinations of values. All these noise placement heuristics inject noise at selected places with a given probability. The probability is set globally for all enabled noise placement heuristics by a *noiseFreq* setting from the range 0 (never) to 1000 (always).

The total number of noise configurations that one can obtain from the above can be computed by multiplying the number of the basic heuristics, which is 6, by 2 reflecting whether *haltOneThread* is/is not used, 2 reflecting whether *timeoutTamper* is used, 100 possible values of noise strength, 5 values of the *sharedVar* heuristics, 3 values of of the coverage-based heuristics, and 1000 values of *noiseFreq*. This gives about 36 million combinations of noise settings. Of course, the state space of the test and noise settings then further grows with the possible values of parameters of test cases and the testing environment.

Test Cases and Test Environment. The experimental results presented in the rest of the paper were obtained using the SearchBestie [19] platform based on the IBM Concurrency Testing Tool (ConTest) [8] and its plug-ins [16, 18] to inject noise into execution of the considered programs and to collect the obtained coverage. The used meta-heuristic algorithms were implemented within the ECJ library [29] which cooperates with the SearchBestie platform as well.

Among our test cases, we include five multi-threaded Java programs used in the previous work on the subject [13]: in particular, the Airlines, Animator, Crawler, Elevator, and Rover case studies (Airlines having 0.3 kLOC, Rover having 5.4 kLOC, and the other programs having around 1.2–1.5 kLOC of code each). Each of these programs contains a concurrency-related error. Moreover, three further multi-threaded benchmark programs, namely, Moldyn, MonteCarlo, and Raytracer, from the Java Grande Benchmark Suite [25], which contain a large number of memory accesses, have been used (their sizes range from 0.8 to 1.5 kLOC). All considered programs have one parametrized test that is used to execute them. All our experiments were conducted on machines with Intel i7 processors with Linux OS, using Oracle JDK 1.6.

4 Objectives and Fitness Function

One can collect various metrics characterizing the execution of concurrent programs. Our testing infrastructure is, in particular, able to report test failures, measure duration of test executions, and collect various code and concurrency coverage metrics [18] as well as numbers of warnings produced by various dynamic analyzers searching for data races [24, 9], atomicity violations [21], and deadlocks [3]. In total, we are able to collect up to 30 different metrics describing concurrent program executions. Collecting all of these data does, of course, introduce a considerable slowdown. Moreover, some of the metrics are more suitable for use as an objective in our context than others.

In this section, we discuss our selection of objectives suitable for solving the TNCS problem through a MOGA-based approach. In particular, we focus on the number of distinctive values produced by the metrics, correlation among the objectives, and their stability. By the stability, we mean an ability of the objective to provide similar values for the same individual despite the scheduling non-determinism. Finally, we introduce a technique that allows us to emphasize uncommon observations and optimize candidate solutions towards testing of such behaviors.

Selection of suitable objectives. It has been discussed in the literature [7] that multi-objective genetic algorithms usually provide the best performance when a relatively low number of objectives is used. Therefore, we try to stay with a few objectives only. Among them, we first include the *execution time of tests* since one of our goals is to optimize towards tests with small resource requirements.

As for the goal of covering as much as possible of (relevant) program behavior, we reflect it in maximizing several chosen concurrency-related metrics. When choosing them, we have first ruled out metrics which suffer from a *lack of distinct values* since meta-heuristics do no work well with such objectives (due to not distinguishing different solutions well enough).

Table 1. Correlation of objectives across all considered test cases.

	Time	*Error*	*WConcurPairs*	*Avio**	*GoldiLockSC**	*GoodLock**
Time	1	-0.083	0.625	-0.036	-0.038	-0.360
Error	-0.083	1	-0.137	-0.213	-0.221	-0.216
WConcurPairs	0.625	-0.137	1	0.116	0.038	-0.263
Avio*	-0.036	-0.213	0.116	1	0.021	-0.274
GoldiLockSC*	-0.038	-0.221	0.038	0.021	1	0.77
GoodLock*	-0.360	-0.216	-0.263	-0.274	0.77	1

Subsequently, we have decided to include some metrics characterizing how well the behavior of the tested programs has been covered from the point of view of finding three *most common concurrency-related errors*, namely, data races, atomicity violations, and deadlocks. For that, we have decided to use the *GoldiLockSC**, *GoodLock**, and *Avio** metrics [18]. These metrics are based on measuring how many internal states of the GoldiLock data race detector [9] or the GoodLock deadlock detector [3], respectively, have been reached, and hence how well the behavior of the SUT was tested for the presence of these errors. The *Avio** metric measures witnessed access interleaving invariants [21] which represent different combinations of read/write operations used when two consecutive accesses to a shared variable are interleaved by an access to the same variable from a different thread. A good point for *GoldiLockSC** and *Avio** is that they usually produces a high number of distinct values [18]. With *GoodLock**, the situation is worse, but since it is the only metric specializing in deadlocks that we are aware of, we have decided to retain it.

Next, in order to account for other errors than data races, atomicity violations or deadlocks, we have decided to add one more metric, this time choosing a general purpose metric capable of producing a high number of distinct tasks. Based on the results presented in [18], we have chosen the *ConcurPair* metric [4] 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 boolean value that is $true$ iff the two locations are visited by different threads. More precisely, we have decided to use the weighted version *WConcurPairs* of this metric [13] which values more coverage tasks comprising a context switch.

Since *correlation among objectives* can decrease efficiency of a MOGA [7], we have examined our selection of objectives from this point view. Table 1 shows the average correlation of the selected metrics for 10,000 executions of our 8 test cases with a random noise setting. One can see that the metrics do not correlate up to two exceptions. *WConcurPairs* and *Time* achieved on average the correlation coefficient of 0.625 and the *GoodLock** and *GoldiLockcSC** metrics the correlation coefficient of 0.77[1]. However, there were also cases where these metrics did not correlate (e.g., the correlation coeffiecient of *GoldiLockSC** and *Avio** was [TODO: 0.021]). We therefore decided to reflect the fact that some of our objectives can sometimes correlate in our choice of a concrete MOGA, i.e., we try to select a MOGA which works well even under such circumstances (cf. Section 5).

[1] In this case, only three of the case studies containing nested locking and hence leading to a non-zero coverage under *GoodLock** were considered in the correlation computation.

Dealing with Scheduling Nondeterminism. Due to the scheduling nondeterminism, values of the above chosen objectives collected from single test runs are unstable. A classic way to improve the stability is to execute the tests repeatedly and use a representative value [15]. However, there are multiple ways how to compute it, and we now aim at selecting the most appropriate way for our setting.

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 of 10 runs. In particular, we considered median (*med*), mode (*mod*)[2], and the cumulative value (*cum*) computed as the sum in the case of time and as the united coverage in case of the considered coverage metrics. We do not consider the often used average value since we realized that the data obtained from our tests were usually not normally distributed, and hence the average would not represent them accurately. We did not consider other more complicated evaluations of representative values due the high computational costs associated with using them. Subsequently, we compared stability of the representative values obtained across the batches. Table 2 shows the average values of variation coefficients of the representatives computed across all the considered configurations for each case study and each way of computing a representative. Clearly, the best average stability was provided by median, which we therefore choose as our means of computing representative values across multiple test runs for all the following experiments.

Table 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

Emphasizing rare observations. When testing concurrent programs, it is usually the case that some behavior is seen very frequently while some behavior is rare. Since it is likely that bugs not discovered by programmers hide in the rare behavior, we have decided to direct the tests more towards such behavior by *penalizing* coverage of frequently seen behaviors. Technically, we implement the penalization as follows. We count how many times each coverage task of the considered metrics got covered in the test runs used to evaluate the first generation of randomly chosen candidate solutions. Each coverage task is then assigned a weight obtained as one minus the ratio of runs in which the task got covered (i.e., a task that was not covered at all is given weight 1, while a task that got covered in 30 % of the test runs is given weight 0.7). These weights are then used when evaluating the coverage obtained by subsequent generations of candidate solutions.

Selected fitness function. To sum up, based on the above described findings, we propose a use of the following fitness functions, which we use in all our subsequent experiments: Each candidate solution is evaluated 10 times, the achieved coverage is penalized, and the median values for the 4 selected metrics (*GoldiLockSC**, *GoodLock**, *WConcurPairs*, and *Time*) are computed.

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

5 Selection of a Multi-objective Optimization Algorithm

Another step needed to apply multi-objective optimization for solving the TNCS problem is to choose a suitable multi-objective optimization algorithm and its parameters. Hence, in this section, we first select one algorithm out of three well-known multi-objective optimization algorithms, namely, *SPEA*, *SPEA2*, and *NSGA-II* [7, 31]. Subsequently, we discuss a suitable setting of parameters of the selected algorithm.

The main role of the multi-objective algorithms is to classify candidate solutions into those worth and not worth further consideration. We aim at selecting one of the algorithms that is most likely to provide a satisfactory classification despite the obstacles that can be faced when solving the TNCS problem. As we have already discussed, these obstacles include the following: (a) Some objectives can sometimes be able to achieve only a *small number of distinct values* because the kind of concurrency-related behavior that they concentrate on does not show up in the given test case. (b) Some of the objectives can sometimes *correlate* as discussed in Section 4. (c) We are working with a *nondeterministic environment* where the evaluation of objectives is not stable. We have proposed ways of reducing the impact of these issues already in Section 4, but we now aim at a further improvement by a selection of a suitable MOGA.

In addition, we also consider the opposite of Issue (a), namely, the fact that some objectives can sometimes achieve rather *high numbers of values*. Dealing with high numbers of values is less problematic than the opposite (since a high number of objective values can be divided into a smaller number of fitness values but not vice versa), yet we would like to assure that the selected MOGA does indeed handle well the high numbers of values and classifies them into a reasonable number of fitness values.

We studied the ability of the considered algorithms to deal with correlation and low or high numbers of distinct objective values using four pairs of objectives. In these pairs, we used the *Avio** metric (based on the Avio atomicity violation detector [21]) and the *GoldiLockSC** metric, which we found to highly correlate with the correlation coefficient of 0.966 in the same kind of correlation experiments as those presented in Section 4 (i.e., they correlate much more than the objectives we have chosen into our fitness functions). As a representative of objectives that often achieve a small number of values, we included the number of detected errors (*Error*) into the experiments, and as a representative of those that can often achieve high values, we take the execution time (*Time*). We performed experiments with 40 different individuals (i.e., test and noise configurations) and evaluated each of them 11 times on the Crawler test case.

Table 3. Pairs of objectives and their evaluation by multi-objective optimization algorithms.

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

Table 3 shows into how many classes the obtained 440 results of the above experiments were classified by the considered algorithms when using four different pairs of objectives. We can see that the *SPEA* algorithm often classifies the results into a very low number of classes while *SPEA2* into a large number of classes, which is close to the number of evaluations. *NSGA-II* stays in all cases in between of the extremes, and we therefore consider it to provide the best results for our needs.

Next, we discuss our choice of suitable values of parameters of the selected MOGA, such as the size of population, number of generations, as well as the selection, crossover, and mutation operators to be used. Our choice is based on the experience with a SOGA-based approach presented in [13] as well as on a set of experiments with *NSGA-II* in our environment. In particular, we experimented with population sizes and the number of generations such that the number of individual evaluations in one experiment remained constant (in particular, 2000 evaluations of individuals per experiment[3]). Therefore, for populations of size 20, 40, and 100, we used sizes of 100, 50, and 20 generations, respectively. Next, we studied the influence of three different crossover operators available in the ECJ toolkit [29] (called *one*, *two* and *any*) and three different probabilities of mutation (0.01, 0.1 and 0.5). As the selection operator, we used the mating scheme selection algorithm instead of the fitness-based tournament or proportional selection which are commonly used in single-objective optimization but provide worse results or are not applicable in multi-objective optimization [14]. We fixed the size of the archive to the size of the population.

In total, we experimented with 27 different settings of the chosen MOGA (3 sizes of population, 3 crossover operators, and 3 mutation probabilities). For each setting, we performed 10 executions of the MOGA process which differ only in the initial random seed values (i.e., only in the individuals generated in the first generation) on the Airlines, Animator, and Crawler test cases. In general, we did not see big differences in the results obtained with different sizes of populations. On the other hand, low probabilities of mutation (0.01 and 0.1) often led to degeneration of the population within a few generations and to low achieved fitness in the last generation. A high mutation set to 0.5 did not suffer from this problem. Further, the *any* crossover operator provided considerably lower values of fitness values in the last generation, while there were no significant differences among the *one* and *two* crossover operators.

Based on these experiments, we decided to work with 50 generations and 20 individuals in a population (i.e., compared with the above experiments, we decrease the number of generations for the given number of individuals since at the beginning the number of coverage tasks grew up, but after the 50th generation there was not much change). In the breeding process, we use the mating scheme algorithm as the selection operator with the recommended parameters $\alpha = 5$, $\beta = 3$; and we use the crossover operator denoted as *two* in ECJ, which takes two selected individuals (integer vectors), divides them into 3 parts at random places, and generates a new candidate solution as the composition of randomly chosen 1st, 2nd, and 3rd part of parents. Finally, to implement mutation, we use an operator that randomly selects an element in the vector of a candidate solution and sets it to a random value within its allowed range with probability 0.5. The resulting agile exploration is compensated by the *NSGA-II* archive, and so the search does not loose promising candidate solutions despite the high mutation rate.

6 Experimental Evaluation

In this section, four experimental comparisons of the proposed MOGA-based approach with the random approach and a SOGA-based approach are presented. First, we show

[3] We used 2000 evaluations because after the 2000 evaluations, saturation used to happen.

that our MOGA-based approach does not suffer from degeneration of the search process identified in the SOGA-based approach in [13]. Then, we show that the proposed penalization does indeed lead to a higher coverage of uncommon behavior. Finally, we focus on a comparison of the MOGA, SOGA, and random approaches with respect to their efficiency and stability. All results presented here were gathered from testing of the 8 test cases introduced in Section 3.

In the experiments, we use the following parameters of the SOGA-based approach taken from [13]: size of population 20, number of generations 50, two different selection operators (tournament among 4 individuals and fitness proportional[4]), 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 build the fitness function of the SOGA-based approach from the objectives selected above[5]:

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

Here, 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 [13], the the SOGA-based approach uses cumulation of results obtained from multiple test runs without any penalization of frequent behaviors.

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

Degeneration of the Search Process. Degeneration, i.e., lack of variability in population, is a common problem of population-based search algorithms. Figure 1 shows average variability of the MOGA-based and SOGA-based approaches computed from the search processes on the 8 considered test cases. The x-axis represents generations, and the y-axis shows numbers of distinct individuals in the generations (max. 20). The higher value the search process achieves the higher variability and therefore low degeneration was achieved. The graph clearly shows that our MOGA-based approach does not suffer from the degeneration problem unlike the SOGA-based approach.

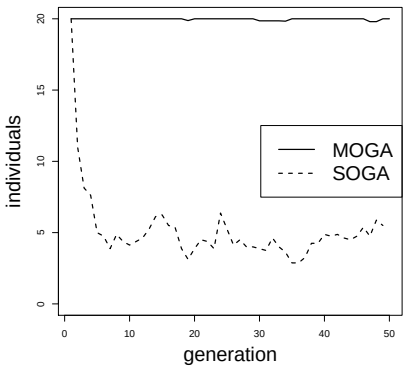

Fig. 1. Degeneration of the MOGA-based and SOGA-based search processes.

[4] Experiments presented in [13] 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.

[5] In the experiments performed in [13], the fitness function was sensitive on weight. Therefore, we remove the weight from our new fitness function for SOGA.

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, but then it basically shifts towards random testing. An interesting observation (probably leading to the good results presented in [13]) 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 behaviors in different executions.

Effect of Penalization. The goal of the above proposed penalization scheme is to increase the number of tested uncommon behaviors. An illustration of the fact that this goal has indeed been achieved is provided in Table 4. The table in particular compares results collected from 10 runs of the final generations of 20 individuals obtained through the MOGA-based and SOGA-based approaches with results obtained from 200 randomly generated individuals. Each value in the table gives the average percentage of uncommon behaviors 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 behaviors that occur often (i.e., in more than 50 % of the runs) while 60 % are rare.

Table 4. Impact of the penalization built into the 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
Moldy	**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

In most of the cases, if some approach achieved the highest percentage of uncommon behaviors under one of the coverage metrics, it achieved the highest numbers under the other metrics as well. Table 4 clearly shows that our MOGA-based approach is able to provide a higher coverage of uncommon behaviors (where errors are more likely to be hidden) than the other considered approaches.

Table 5. 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
Moldy	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 SOGA-based approaches and 200 test runs under random generated test and noise settings. Table 5 compares the

efficiency of these tests. In order 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 time unit (milisecond). Next, *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 the considered ways. The last row gives the average improvement (*Avg. impr.*) of the genetic approaches against random testing. We can see that both genetic approaches are significantly better than the random approach. In some cases, the MOGA-based approach got better evaluated while SOGA won in some other cases. Note, however, that as shown in the previous paragraph, the MOGA-based approach is more likely to cover rare tasks, and so even if it covers a comparable number of tasks with the SOGA-based approach, it is still likely to be more advantageous from the practical point of view.

Stability of Testing. Finally, we show that candidate solutions found by our MOGA-based approach provide more stable results than the SOGA-based and random approaches. In particular, for the MOGA-based and SOGA-based approaches, Table 6 gives 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 6. 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

7 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 the *internal validity*, i.e., whether there has been a bias in the experimental design or stochastic behavior of the meta-heuristic search algorithms that could affect the obtained results. In order to deal with this issue, Section 5 provides a brief discussion and experimental evidence that supports the choice of the NSGA-II MOGA algorithm out of the three considered algorithms. In order to address the problem of setting the various parameters of meta-heuristic algorithms, a number of experiments was conducted to choose configurations that provide good results in the given context. Similarly, our choice of suitable objectives was done based on observations from previous experimentation [18]. Care was taken to ensure that all approaches are evaluated in the same environment.

Another issue to address is that of the *external validity*, i.e., whether there has been a bias caused by external entities such as the selected case studies (i.e., 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 in order to confirm the results presented here.

8 Conclusions and Future Work

In this paper, we proposed an application of multi-objective genetic optimization for finding good settings of tests and noise. We discussed a selection of suitable objectives taking into account their usefulness for efficiently finding concurrency-related errors as well as their properties important for the process of genetic optimization (numbers of distinct values, correlation) as well as for stability. We also proposed a way how to emphasize uncommon behaviors in which so-far undiscovered bugs are more likely to be hidden. Further, we compared suitability of three popular multi-objective genetic algorithms for our purposes, which showed that the NSGA-II algorithm provides the best ability to classify candidate solutions in our setting. Finally, we demonstrated on a set of experiments with 8 case studies that our approach does not suffer from the degeneration problem, it emphasizes uncommon behaviors, and generates settings of tests and noise that improve the efficiency and stability of the testing process.

As a part of our future work, we plan to further improve the efficiency and stability of the generated test and noise settings. For this purpose, we would like to exploit the recently published results [10] indicating that searching for a test suite provides better results than searching for a set of the best individuals.

Acknowledgement. We thank Shmuel Ur and Zeev Volkovich for many valuable comments on the work presented in this paper. The work was supported by the Czech Ministry of Education under the Kontakt II project LH13265, the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT projects FIT-S-12-1 and FIT-S-14-2486. Z. Letko was funded through the EU/Czech Interdisciplinary Excellence Research Teams Establishment project (CZ.1.07/2.3.00/30.0005).

References

1. E. Alba and F. Chicano. Finding Safety Errors with ACO. In *Proc. of GECCO'07*, ACM, 2007.
2. N. Ayewah, W. Pugh, J. D. Morgenthaler, J. Penix, and Y. Zhou. Using FindBugs on Production Software. In *Proc. of OOPSLA'07*, ACM, 2007.
3. S. Bensalem and K. Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of PADTAD'05*, LNCS 3875, Springer-Verlag, 2005.
4. A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM, 2005.
5. H. Chockler, E. Farchi, B. Godlin, and S. Novikov. Cross-entropy Based Testing. In *Proc. of FMCAD '07*, IEEE, 2007.
6. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.
7. K. Deb. *Multi-Objective Optimization Using Evolutionary Algorithms*. Wiley paperback series. Wiley, 2009.

8. 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), John Wiley & Sons, Ltd., 2003.
9. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, ACM, 2007. ACM.
10. G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2), 2013.
11. P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.
12. 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.
13. V. Hrubá, B. Křena, Z. Letko, S. Ur, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*, LNCS 7515, Springer-Verlag, 2012.
14. H. Ishibuchi and Y. Shibata. A Similarity-based Mating Scheme for Evolutionary Multiobjective Optimization. In *Proc. of GECCO'03*, LNCS 2723, Springer, 2003.
15. Y. Jin and J. Branke. Evolutionary Optimization in Uncertain Environments – A Survey. *IEEE Transactions on Evolutionary Computation*, 9(3), 2005.
16. 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.
17. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-the-fly. In *Proc. of PADTAD'07*, ACM, 2007.
18. B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*, LNCS 7186, Springer-Verlag, 2012.
19. 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.
20. B. Křena, Z. Letko, and T. Vojnar. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*, LNCS 7119, Springer-Verlag, 2012.
21. S. Lu, J. Tucek, F. Qin, and Y. Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*, ACM, 2006.
22. M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *OSDI*, USENIX Association, 2008.
23. T. Peierls, B. Goetz, J. Bloch, J. Bowbeer, D. Lea, and D. Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
24. 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.
25. L.A. Smith, J.M. Bull, J. Obdržálek. A Parallel Java Grande Benchmark Suite. In *Proc. of Supercomputing'01*, ACM, 2001.
26. A. Spillner, T. Linz, and H. Schaefer. *Software Testing Foundations: A Study Guide for the Certified Tester Exam*. Rocky Nook, 3rd edition, 2011.
27. J. Staunton and J. A. Clark. Searching for Safety Violations Using Estimation of Distribution Algorithms. In *Proc. of ICSTW'10*, IEEE, 2010.
28. S. Steenbuck and G. Fraser. Generating Unit Tests for Concurrent Classes. In *ICST'13*, IEEE, 2013.
29. D. White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, 13, 2012.
30. 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.
31. E. Zitzler. *Evolutionary Algorithms for Multiobjective Optimization: Methods and Applications*. PhD thesis, ETH Zurich, 1999.