Testing of Concurrent Programs Using Genetic Algorithms

Vendula Hrubá[1], Bohuslav Křena[1], Zdeněk Letko[1], Shmuel Ur[2], and Tomáš Vojnar[1]

[1] IT4Innovations Centre of Excellence, FIT, Brno University of Technology, Czech Republic
{ihruba, krena, iletko, vojnar}@fit.vutbr.cz
[2] Shmuel Ur Innovations, Ltd., shmuel.ur@gmail.com

Abstract. Noise injection disturbs the scheduling of program threads in order to increase the probability that more of their different legal interleavings occur during the testing process. However, there exist many different types of noise heuristics with many different parameters that are not easy to set such that noise injection is really efficient. In this paper, we propose a new way of using genetic algorithms to search for suitable types of noise heuristics and their parameters. This task is formalized as the test and noise configuration search problem in the paper, followed by a discussion of how to represent instances of this problem for genetic algorithms, which objectives functions to use, as well as how to set parameters of genetic algorithms when solving the problem. The proposed approach is evaluated on a set of benchmarks, showing that it provides significantly better results than the so far preferred random noise injection.

1 Introduction

The arrival of multi-core processors into common computers accelerated development of software with multi-threaded design. Multi-threaded programming is, however, significantly more demanding and offers much more space for errors. Moreover, errors in concurrency are often very difficult to discover and localise due to the non-deterministic nature of multi-threaded computation. This situation stimulates research efforts devoted to all sorts of methods for testing, analysis, and verification.

Formal methods of verification, such as, e.g., model checking [11], aim at precise program verification. Unfortunately, these precise approaches do not scale well for complex software. This is one of the main reasons why heuristic approaches such as lightweight static analyses, testing, and dynamic analyses are still very popular.

When dealing with concurrent programs, testing and dynamic analysis that rely on executing the program under test and evaluating the witnessed run suffer from the problem of non-deterministic scheduling of program threads. Due to this problem, a single execution of a program is insufficient to determine correctness of 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 the same input will produce an incorrect result.

One way to address this problem is to use *deterministic testing* that can be viewed as model checking bounded in various ways (e.g., in the number of context switches) [18, 21]. This technique attempts to systematically test all interleaving scenarios up to some bound, which is quite demanding (especially for long runs) because one needs to track which scheduling scenarios have been witnessed and systematically force new ones.

A lightweight alternative to the above is to use *noise injection techniques* [7] based on heuristically disturbing the scheduling of program threads in hope of observing so far unseen scheduling scenarios. Although this approach cannot prove correctness of a program even under some bounds on its behaviour, it was demonstrated in [7, 15] that it can rapidly increase the probability of spotting concurrency errors.

Noise injection can be implemented by instrumenting the program under test by noise generation code that influences the execution of selected threads at selected program locations. Noise injection can use different *noise seeding heuristics* given by the *type of noise* (e.g., noise based on injecting calls of `yield()`, calls of `wait()`, halting one thread till other threads can continue, etc.), the *strength of the noise* (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.), and the *frequency of the noise* (how often some noise is generated at a particular location). Moreover, various *noise placement heuristics* can be used, including the use of a fixed set of program locations at which some concurrency-relevant actions appear (such as accesses to shared memory, synchronisation, etc.), using a randomly selected subset of such a set, or some more involved heuristics driven by the so-far obtained coverage of the program behaviour [15].

Our previous work on noise injection [15] shows that there is no silver bullet among the many existing noise injection heuristics. Results provided by them depend on the tested program as well as on the run-time environment (the type and number of processors and the actual workload are usually the most significant factors). Actually, some configurations can decrease the probability of an error manifestation. This is helpful for run-time healing of errors [13], but it is highly undesirable for detecting them. Moreover, the number of possible settings of the noise injection (and also of the test itself) together with the considerable time needed to run a test in order to evaluate the efficiency of a certain noise configuration makes exhaustive searching for suitable noise configurations impractical. This is exactly the case where *metaheuristic search techniques* [19] can help.

Genetic algorithms [19] are metaheuristic search techniques which try to find the best solutions by sampling the search space. They start with an initial set (called a *generation*) of possible solutions (also called *individuals*). Each individual is evaluated and assigned a value called a *fitness* representing the suitability of the solution it represents. The next generation of individuals is typically obtained by a stochastic recombination (called a *crossover*) and *mutation* of individuals selected according to their fitness.

In this paper, we propose a new way of using genetic algorithms to search for suitable types of noise heuristics and their parameters. We formalize this task as the *test and noise configuration search problem* (the TNCS problem). Then, we show how to represent instances of this problem for genetic algorithms, and we discuss which basic objective functions may be useful as building blocks of complex objective functions suitable in the given context. We also discuss how to set parameters of genetic algorithms when solving the TNCS problem. Next, we instantiate the framework by a concrete combined objective function suitable especially (but, as our experiments show, not only) for data race detection. Finally, we evaluate the proposed approach on a set of benchmarks, showing that it provides significantly better results than the so far preferred random noise injection.

Plan of the paper. The rest of the paper is organised as follows. In Section 2, we discuss the related work. In Section 3, we formulate the TNCS problem and discuss the basic objective functions suitable in its context. In Section 4, propose how to utilise genetic algorithms to solve the TNCS problem. Section 5 focuses on setting parameters of genetic algorithms for the TNCS problem. In Section 6, we propose a concrete combined objective function for use with the TNCS problem, and we provide experimental evidence on how the proposed approach can improve the testing process. Finally, Section 7 provides concluding remarks and comments on the possible future work.

2 Related Work

Most existing works in the area of search-based testing of concurrent programs focus on applying various metaheuristic techniques to control the state space exploration within the *guided model checking* approach [11]. The basic idea is to explore areas of the state space that are more likely to contain concurrency errors even when the entire state space will not be explored. Metaheuristic algorithms that have been applied within the guided model checking approach for finding deadlocks and/or assertion violations include simulated annealing [6], genetic algorithms [11, 3], the partial swarm optimisation (PSO) [6], and the ant colony optimisation (ACO) [1, 2]. An advantage of this approach is that the underlying model checking offers a well-defined state space and a high degree of determinism. On the other hand, the approach shares limitations of model checking in terms of scalability and cost of modelling of the environment. In our approach, we focus on testing and dynamic analyses which are able to handle much larger real programs.

In [9], genetic algorithms are applied within the process of debugging of concurrent programs based on repeated testing with noise injection. Genetic algorithms are used in order to find a noise configuration which causes concurrency-related bugs to occur with a high probability while preferring settings with noise concentrated to a minimal number of locations (which is motivated by concentrating on the problem of debugging). Compared to [9], we do not search for which concrete locations should be noised with which noise. Instead, we search for which noise seeding and noise placement heuristics (or which combinations of these heuristics) with which parameters can provide good results for a particular test and environment. This allows us to use a simpler representation of individuals and to support much larger test cases with plenty of possible locations to be noised. Moreover, we propose new objective functions (based, e.g., on results that can be obtained through various dynamic analyses) which allow us to focus not only on debugging but also on testing. Finally, compared with the initial results presented in [9], we present a more thorough experimental evaluation.

The problem of increasing the probability of an error manifestation within the debugging process is targeted in [4, 20] too. However, these works do not consider metaheuristic search algorithms. In [4], program locations are first statically classified according to their suitability for noise injection. Then, a probabilistic algorithm is used to find a subset of program locations that increase the error manifestation ratio. In [20], a machine learning feature selection algorithm is used to identify a subset of program locations where to inject noise. In this case, the test is executed many times, and program locations where the noise was injected in each execution are collected together with information whether the error got manifested.

Finally, in [16], we presented our preliminary results with a steepest ascending search algorithm. The experiments proved our concept but showed that the local search technique is not suitable for the given setting. The used algorithm showed a tendency to find a local optimum only. Therefore, in this work, we focus on global search techniques, namely, genetic algorithms.

3 Testing of Concurrent Programs as a Search Problem

In this section, we present our proposal of how search techniques can be combined with noise-based testing of concurrent programs by identifying suitable combinations of noise injection heuristics as well as their parameters. In particulate, we formulate the proposed use of search techniques via the so-called test and noise configuration search (TNCS) problem. Subsequently, we discuss several objective functions that can be useful when dealing with various instances of the TNCS problem (typically as building blocks of more complex combined objective functions as we illustrate in Section 6).

3.1 The Test and Noise Configuration Search Problem

As we have mentioned already in the introduction, there are two main issues that must be solved when using noise injection. First, one needs to determine program locations where to insert noise. Heuristics which target this problem are called *noise placement heuristics*. Second, one needs to determine which *noise seeding heuristics*, i.e., which way of disturbing thread scheduling, should be used. Moreover, most types of the heuristics are adjustable by one or more parameters influencing their behaviour and efficiency (e.g., noise seeding heuristics are often parameterized by their strength). Further, one can combine several noise placement and seeding techniques within one execution. Indeed, our results presented in [15] show that such a combination provides in many cases better results than using a single heuristics. Finally, it is usually the case that there exist multiple test cases for a given program that can also be parametric.

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

Formally, let $Type_P$ be a set of available types of noise placement heuristics each of which we assume to be parameterized by a vector of parameters. Let $Param_P$ be a set of all possible vectors of parameters. Further, let $P \subseteq Type_P \times Param_P$ be a set of all allowed combinations of types of noise placement heuristics and their parameters. 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*.

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

3.2 Objective Functions for the Context of the TNCS Problem

We next suggest several possible objective functions that can be useful in various instances of the TNCS problem, typically combined into more complex objective functions as we illustrate in Section 6.

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

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

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

A further possibility is to use a suitable *coverage metric* allowing one to judge how much of the possible behaviour of the program under test has been covered (and hence how likely it is that some undesired behaviour was omitted) and to look for test configurations maximising the obtained coverage. Concurrency-related metrics based on dynamic analyses which we presented in [14] can be especially useful here. These metrics are not based on simply counting the number of produced warnings, but on much finer measures. Some of them are based on monitoring events that make the internal state of a dynamic analyser change, e.g., the *HBPair* metrics based on the happens-before relations, and some express how many internal states a certain dynamic analyser reached, e.g., the *GolidLockSC* metrics based on monitoring the internal states of the GoldiLock analyser [8]. Of course, there are many other existing coverage metrics which can be considered. For instance, the synchronisation coverage [5] (*Synchro*) which measures how well the various synchronisation mechanisms used in the program under test are tested (by measuring how many different scenarios of the use of the synchronisation mechanisms were witnessed). A drawback of many concurrency coverage metrics is that it is often impossible to compute what the full coverage is; this is, however, not

a problem here since we are interested in relative comparisons of the coverage achieved through different test configurations.

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

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

4 A Genetic Approach to the TNCS Problem

In this section, we present our proposal of using a genetic approach to solving the TNCS problem. The approach is presented on a concretization of the TNCS problem for the context of using the IBM Concurrency testing tool (ConTest) [7] (with some extensions) for noise-based testing. We therefore start by presenting the concrete set of noise configurations considered. Subsequently, we present how one can apply the genetic approach in this concrete setting.

4.1 ConTest-based Noise Configurations

We consider noise injection heuristics implemented in ConTest extended by our plug-in implementing a coverage-based noise placement heuristics [15]. Hence, we consider three noise placement heuristics: the *random* heuristics which picks program locations randomly, the *sharedVar* heuristics which focuses on accesses to shared variables, and our *coverage-based* heuristics [15] which focuses on accesses near a previously detected thread context switch. The *sharedVar* heuristics has two parameters modifying its behaviour with 5 valid combinations of its values. The *coverage-based* heuristics is controlled by 2 parameters with 3 valid combinations of values. 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 *random* heuristics is enabled by default when $noiseFreq > 0$. The *random* heuristics can be suppressed by one parameter of the *sharedVar* heuristics which explicitly disables the combination of these two heuristics.

The considered infrastructure offers 6 basic and 2 advanced noise seeding techniques. 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* and *synchYield* techniques lock a special

monitor and then either wait for some time or call `yield()`. 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. 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).

4.2 Individuals, Their Encoding, and Genetic Operations on Them

In order to utilise a genetic algorithm to solve the TNCS problem with the considered set of noise configurations, we let the particular test configurations play the role of *individuals*. We encode the test configurations as *vectors of integers*. The test configuration is either reduced to solely a noise configuration (when a single test case without parameters is considered), or it consists of the noise configuration extended by one or more specific entries controlling the test case settings. We, however, concentrate here on the noise configurations only, which form vectors of numbers in the range $(0, 0, 0, 0, 0, 0)$–$(1000, 5, 3, 6, 2, 2)$. Here, the first entry controls the *noiseFreq* setting, the next two control the *sharedVar* and *coverage-based* noise placement heuristics. The last three entries control the setting of the basic and advanced noise seeding heuristics. Each entry in the vector is annotated by a flag saying whether there exists an ordering on the values of the entry. We call entries whose values are ordered as *ordinal entries*.

We consider the standard one-point, two-point, and uniform element-wise (any-point) *crossover operators* [19] in the form they are implemented in the ECJ library [22]. *Mutation* is also done on an element-wise basis, and it handles ordinal and non-ordinal entries differently. Non-ordinal entries are set to a randomly chosen value from the particular range (including the current value). Ordinal entries (e.g., entries encoding the strength of noise or the parameter controlling the number of threads the test should use) are handled using the standard Gaussian mutation [19] (with the standard deviation set to 10 % of the possible range or minimal value 2). Finally, we consider standard proportional and tournament-based fitness selection operators [19] as they are implemented in the ECJ library.

5 Parameters of Genetic Algorithms and the TNCS Problem

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

5.1 Experiments for Finding Suitable Parameters of Genetic Algorithms

We conducted all our experiments aimed at finding a suitable setting of the parameters of genetic algorithms on one selected case study only. This is mostly due to the high

time consumption of evaluating each test configuration through multiple test executions. In particular, we used the *Crawler* test case which is based on a part of an older version of a major IBM production software. The crawler creates a set of threads waiting for a connection. If a connection simulated by the testing environment is established, a worker thread serves it. There is an error in a method that is called when the crawler shuts down. The error causes an unhandled exception. The trickiness of the error can be seen from the very low probability of its manifestation (approximately 0.0006 when no noise is used). The case study has 19 classes and 1.2 kLOC. There is a single test case available with no parameters (making test configurations equal to noise configurations).

We conducted our experiments on multiple machines, all having Intel i5 661 processors, running 64-bit Linux and Java 6. We used our infrastructure SearchBestie [16] and IBM ConTest to evaluate test configurations and the ECJ library [22] to implement the genetic algorithms. We narrowed the search space down by sampling the *noiseFreq* parameter by ten, i.e., by reducing its possible values to 0, 10, ..., 1000.

With the aim of observing as many behaviours differing in their various important concurrency-related aspects as possible, we considered an objective function maximising the obtained coverage under three different concurrency coverage metrics, namely, *Synchro*, *Avio** and *HBPair** [14]. This objective function covers three different aspects of concurrency behaviour: interleaving of accesses from different threads to shared memory locations via *Avio**, successful synchronisation of program threads inducing a happens-before relation via *HBPair**, and information about whether the implemented synchronisation does something helpful via *Synchro*. We used results of approximately 1 million randomly noised executions to estimate the 100 % achievable coverage (denoted as max below) for each of the metrics and set up the following fitness function:

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

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

We fixed the number of evaluated individuals to 2000 in each experiment. According to our experiments, this value is sufficient to reach saturation of the selected coverage metrics in the Crawler case study. We set the size of the considered populations and number of generations as follows (population/generation size): 200/10, 80/25, 40/50, 20/100, and 10/200. We considered the breeding infrastructure to consist of two selection operators which select individuals for the crossover operator. The output of the crossover operator was mutated using the mutation operator described in Section 4.2.

We performed two sets of experiments. In the first one, we considered the standard fitness proportional selection operators, four different standard crossover operators (*one-point*, *two-point*, and *any-point* with the probability of mutating each element of the vector set to 0.1 and 0.25), and four different probabilities of applying the mutation operator (0.01, 0.1, 0.25, and 0.5). In the second set of experiments, we fixed the considered size of the population to 40 and the number of generations to 50, the crossover operator to *any-point* with probability 0.1, and the mutation probability to 0.01. We then studied the influence of different selection operators, elitism which puts into the next

generation a number of individuals (0, 2, 4) without breeding, and a random creation of a few individuals (0, 2, 4) that are put into the following generation. We considered the fitness proportional and tournament selection operators (with the size of the tournament being 2 or 4) and their combinations.

From each experiment, we collected various data concerning the generated populations including, in particular, the following two statistics: (1) The average fitness value in each generation *aver* and (2) the best individual fitness in each generation *best*. Our goal was then to identify parameters of the genetic algorithms under which the best test configuration out of all discovered test configurations is found, and it is found as quickly as possible. For that, we used the *best* and *aver* statistics. The results of the experiments are summarised below with some more technical details available in [12].

5.2 Results of Experiments with the Parameters of Genetic Algorithms

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

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

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

Based on the results summarised above, we found as suitable the following setting of the parameters of genetic algorithms for the considered concretisation of the TNCS problem: Size of population 20, two different selection operators (tournament among 4 individuals and fitness proportional), the *any-point* crossover with a higher probability (0.25), a low mutation probability (0.01), and two elites (that is 10 % of the population). This parameter setting is used in the experiments presented in the next section.

6 A Concrete Application of the Proposed Approach

In this section, we first propose a complex objective function for the TNCS problem that carefully combines the above discussed basic objective functions, finally leading to a concrete application of genetic algorithms for improving the process of testing of

concurrent programs. In particular, the stress is on looking for data races, but as our experiments show, the approach helps in finding other kinds of concurrency-related errors too. Next, we present a collection of benchmarks and results of experiments with them which illustrate the efficiency of our approach.

6.1 A GoldiLocks-based Objective Function

Based on our experience with different concurrency coverage metrics and dynamic error detectors, we have decided to build our concrete objective function on maximizing the coverage obtained under the concurrency coverage metric *GoldiLockSC* [14] based on the GoldiLocks algorithm [8], together with maximizing the number of actual warnings produced by this algorithm. We have chosen the GoldiLocks algorithm for our objective function because it has a low ratio of false positives, and it is able to continue in the analysis even after an error is detected. Moreover, our results indicate that the concurrency coverage metric *GoldiLockSC* has multiple positive properties. In particular, the coverage under this metric usually grows smoothly (i.e., with a minimum of shoulders) and does not stabilise too early (i.e., before most behaviours relevant from the point of view of data race detection are examined). Further, based on the discussion presented in Section 3.2, we also reflect in our objective function an intention to minimize the execution time and to maximize the number of detected errors.

In summary, we thus aim at (1) maximizing coverage under the concurrency coverage metric *GoldiLockSC* [14], (2) maximizing the number of warnings produced by GoldiLocks, (3) maximizing the number of detected real errors due to data races, and (4) minimizing the execution time. The different basic objectives are combined using a system of weights assigned to them.

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

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

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

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

6.2 Case Studies

We concentrate primarily on data race detection, but we also try to apply our genetic approach to case studies containing other kinds of concurrency errors (and, as we will show, we obtain quite positive results even in such cases). In particular, we evaluate our approach on 5 test cases containing concurrency-related errors. The test cases are listed in Table 1. In the table, the *kLOC* column shows the size of the considered test case, and the *Param* column indicates the number of its parameters and the number of possible values of each parameter (e.g., $2, 2$ means that the test takes two parameters, each with two possible values).

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

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

The *Airlines* and *Animator* test cases were run on Intel Core2 6600 machines, the *Rover* test case on a machine with an Intel i5-2500 processor, and the *FTPServer* test case on a machine with two Intel X5355 processors. In case of the *Crawler* test case, two different hardware environments were used. The first (denoted simply as *Crawler* in Table 1) used a machine with an Intel i5 661 processor, while the second (denoted *Crawler**) was executed on a machine with four AMD Opteron 8389 processors. These two options were used on purpose in order to study how our approach works in different hardware environments. All mentioned computers ran 64bit Linux and Java version 6. A more detailed description of the test cases and their parameters can be found in [12].

6.3 Experimental Results

To evaluate the efficiency of our approach when using the GoldiLocks-based objective function, we again used the infrastructure described in Section 5. We use the setting of parameters of genetic algorithms inferred in Section 5. Although this setting was inferred for a different objective function and using sampled values of the *noiseFreq* parameter only, we believe that it represents a good option even for other experiments with our genetic algorithm. Indeed, the objective function used in Section 5 was designed to be rather general in order to cover a lot of different concurrent behaviours. Moreover,

Table 1. An experimental comparison of the proposed genetic approach with the random approach to setting test and noise parameters

Test case			Best configuration			Breeding process		
Name	kLOC	Params	Gen.	Error	Time	Error	Error*	Time
Airlines	0.3	5,5,10	15	3.0 / 1.7	3.8 / 2.5	3.2	8.8	3.0
Animator	1.5	–	25	21.8 / 10.9	1.1 / 1.3	4.3	5.4	1.3
Crawler	1.2	–	22	– / –	1.3 / 1.5	0.3	1.1	3.3
Crawler*	1.2	–	25	– / –	1.1 / 1.1	0.4	1.0	2.8
FTPServer	12.2	10	14	1.2 / 1.0	3.8 / 4.7	0.9	1.7	1.9
Rover	5.4	7	3	★	33.7 / 19.4	3.2	8.8	3.0

we analysed the correlation between the values of the fitness function of Section 5 and the $GoldiLocksSC$ metric used in the GoldiLocks-based objective function on the performed experiments and realized that the correlation is high. After all, the combination of *HBPair** and *Avio** focuses on the same events as the GoldiLocks algorithm.

In the experiments, we allowed the elite individuals to be re-evaluated in the following generations. This is motivated by the fact that a few executions of an individual (5 in our case) are often not sufficient to prove whether the configuration can make a concurrency error manifest. Indeed, tricky concurrency-related errors manifest very rarely even if a suitable noise heuristics is used [15]. The reevaluation of elites therefore gives the most promising individuals another chance to spot an error. This setting is a compromise between a high number of executions needed to evaluate every individual more times and the available time we have.

We compare our genetic approach with the random approach to the choice of noise heuristics and their parameters. In the random approach, we randomly select 2000 test and noise configurations and let our infrastructure evaluate them in the same way we evaluate individuals in the genetic approach. Table 1 summarises our results. The table is based on average results obtained from 10 executions of the genetic and random approach. It is divided into three parts. In the left part (*Test case*), the test cases are identified, and their size and information about their parameters are provided.

An Evaluation of the Best Individuals. The middle part of Table 1 (*Best individual*) contains three columns which compare the best individual obtained by our genetic approach and found by the random approach. The *Gen.* column contains the average number of generations (denoted as gen below) within which we discovered the best individual according to the considered fitness function. The numbers indicate that we are able to find the best individual according to the considered fitness function within the first quarter of the considered generations. This motivates our future work to design a suitable termination condition for our specific testing process.

The *Error* column of the *Best individual* section of Table 1 compares the ability of the best individual to detect an error. The column contains two values (x_1/y_1). The first value x_1 is computed as the fraction of the average number of errors found by the best individual computed by the genetic algorithm and the average number of errors discovered by the best individual found by the random generation provided that an equivalent number of executions is provided to the random approach (this number is computed as gen times the size of the population which is 20). The second number y_1 is computed as the fraction of the average number of errors found by the best individual computed by the genetic algorithm and the average number of errors discovered by the best indi-

vidual found randomly in 2000 evaluations. The –/– value represents a situation where none of the best individuals was able to detect the error within the allowed 5 executions. The ★ symbol means that the genetically obtained best individual did not spot any error while the best individual found by the random generation did (we discuss this situation in more detail below).

Similarly, the *Time* column of the *Best individual* section of Table 1 compares average times needed to evaluate the best individual obtained by our approach and the best individual found by the random approach. Again, two values are presented (x_2/y_2). The first value x_2 is computed as the average time needed by the best individual found by the random approach if only $gen * 20$ evaluations are considered, divided by the average time the genetically found best individual needed. The second value y_2 shows the average time needed by the best individual found by the random generation when it was provided with 2000 evaluations, divided by the average time needed by the genetically found best individual.

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

An Evaluation of The Breeding Process. The right part of Table 1 (*Breeding process*) provides a different point of view on our results. In this case, we are not interested in just one best individual learned genetically or by random generation that is assumed to be subsequently used in debugging or regression testing. Instead, we focus on the results obtained during the breeding process itself. The genetic algorithm is hence considered here to play a role of a heuristics that directly controls which test and noise configurations should be used during a testing process with a limited number of evaluations that can be done (2000 in our case).

This part of the table contains three columns which compare the genetic and random approaches wrt. their successes in finding errors and wrt. the time needed to perform the 2000 evaluations. The first column (*Error*) compares the average number of errors spot during the breeding process and the average number of errors spot during the evaluation of 2000 randomly chosen configurations of the test and noise heuristics. The *Error** column compares the average number of errors detected by our genetic approach with the average number of errors spot by the random approach when the random approach

is provided with the same amount of time as the genetic approach. Finally, the *Time* column compares the average total time needed by the random approach in 2000 evaluations and the average time needed by our genetic approach. Again, the values higher than 1 in all the columns represents how many times our approach outperforms the random approach.

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

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

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

7 Conclusions and Future Work

In this work, we have formulated the test and noise configuration search (TNCS) problem, and we have proposed a way how to use genetic algorithms to solve it. We have performed experiments aimed at choosing suitable parameters of genetic algorithms to be used when solving the problem. We have instantiated the framework for the case of noise injection techniques implemented in the ConTest tool and its extensions and proposed a complex objective function suitable when aiming at data race detection (but successful even when looking for other kinds of bugs). We have performed experiments on a set of benchmark programs showing that our approach significantly outperforms the commonly used approach of randomly selecting noise configurations.

The proposed approach can be further improved, e.g., by development of termination conditions which would help one to determine when to stop the breeding process. Other interesting subjects for future work include development of improved objective functions, e.g., objective functions eliminating the negative effect we saw in the *Rover* test case, objective functions which focus on different kinds of concurrency errors (e.g., deadlocks), or which are able to focus on corner cases of the tests.

Acknowledgement. This work was supported by the Czech Science Foundation (projects P103/10/0306 and 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070, and the internal BUT projects FIT-11-1 and FIT-12-1.

References

1. E. Alba and F. Chicano. Finding Safety Errors with ACO. In *Proc. of GECCO'07*, ACM Press, 2007.
2. E. Alba and F. Chicano. Searching for Liveness Property Violations in Concurrent Systems with ACO. In *Proc. of GECCO'08*, ACM Press, 2008.
3. E. Alba, F. Chicano, M. Ferreira, and J. Gomez-Pulido. Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms. In *Proc. of GECCO'08*, ACM Press, 2008.
4. Y. Ben-Asher, Y. Eytani, E. Farchi, and S. Ur. Noise Makers Need To Know Where To Be Silent–Producing Schedules That Find Bugs. In *Proc. of ISOLA'06*, IEEE CS, 2006.
5. A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM Press, 2005.
6. F. Chicano, M. Ferreira, and E. Alba. Comparing Metaheuristic Algorithms for Error Detection in Java Programs. In *Proc. of SSBSE'11*, LNCS 6956, Springer, 2011.
7. O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for Testing Multi-Threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, Wiley, 2003.
8. T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*, ACM Press, 2007.
9. Y. Eytani. Concurrent Java Test Generation as a Search Problem. *ENTCS*, 144:57–72, Elsevier, 2006.
10. D. Giannakopoulou, C. S. Pasareanu, M. Lowry, and R. Washington. Lifecycle Verification of the NASA Ames K9 Rover Executive. In *Proc. of ICAPS'05*, 2005.
11. P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. *STTT*, 6(2):117–127, Springer, 2004.
12. V. Hrubá, B. Křena, Z. Letko, and T. Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. Technical report, FIT BUT, 2012.
    Available at `http://www.fit.vutbr.cz/~iletko/pub/tr-2012-01.pdf`.
13. B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. Healing Data Races On-the-Fly. In *Proc. of PADTAD'07*, ACM Press, 2007.
14. 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, 2012.
15. B. Křena, Z. Letko, and T. Vojnar. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'12*, LNCS 7119, Springer, 2012.
16. B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*, ACM Press, 2010.
17. Z. Letko. Sophisticated Testing of Concurrent Software. In *Proc. of SSBSE'10*, IEEE CS, 2010.
18. M. Musuvathi, S. Qadeer, and T. Ball. Chess: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.
19. E.-G. Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.
20. R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*, ACM Press, 2007.
21. J. Šimša, R. Bryant, and G. Gibson. dBug: Systematic Testing of Unmodified Distributed and Multi-threaded Systems. In *Proc. of SPIN'11*, LNCS 6823, Springer, 2011.
22. D. White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, 13:65–67, Springer, 2012.