Analysis and Testing of Concurrent Programs

Zdeněk Letko*
Faculty of Information Technology
Brno University of Technology
Božetěchova 1/2, 613 00 Brno, Czech Rep.
iletko@fit.vutbr.cz

Abstract

In this paper, a methodology for deriving concurrency coverage metrics which measure how well the synchronisation and concurrency-related behaviour of tested programs has been examined is introduced. Next, our experiences with testing multi-threaded programs using a noise injection technique are discussed showing that there is no silver bullet among the noise injection techniques. Finally, a novel use of stochastic optimisation algorithms in the area of concurrency testing is proposed in the form of their application for finding suitable combinations of values of the many parameters of tests and the noise injection techniques. The approach has been implemented in a prototype way and tested on a set of benchmark programs, showing its potential to significantly improve the testing process.

1. Introduction

The arrival of multi-core processors into regular computers accelerated development of software that uses multi-threaded design to utilize the available hardware resources. Threads which exist within a process share process resources such as process memory which makes communication among threads seemingly easier but, on the other hand, prone to errors. Errors in concurrency are not only easy to cause, but also very difficult to discover and localize due to the non-deterministic nature of multi-threaded computation. This situation stimulates research efforts which are currently devoted to all sorts of methods for discovering errors in concurrency (or, for proving their absence), including testing, dynamic analysis, as well as various approaches of formal verification.

Program testing is the most common way of finding errors in programs. In testing, a programmer or tester creates a *test case* which is defined by inputs and corresponding outputs. The test case is executed. If a failure occurs, there is an error in the program or in the test case. Testing is often combined with *coverage analysis*—a process of collecting, reviewing and analysing *coverage metrics* which allow to measure occurrence of a considered phenomena during the testing. *Dynamic analysis* which is also often called *runtime verification* is based on *program tracing* in which he gathered information from the test execution is analysed with an intention to discover abnormal execution conditions. Techniques that do not execute the program include static analysis, theorem proving, abstract interpretation and model checking. There exist many different approaches ranging from rather simple code patterns static analyses to quite complex formal automatic or semi-automatic methods such as abstract interpretation.

Despite the intense research in the area, deterministic testing, advanced static analyses, abstract interpretation, and model checking which are able to prove correctness of multi-threaded programs are still too demanding and do not scale well. Instead, simple static analyses, non-deterministic testing, and dynamic analysis are usually used by software developers and testers to search for errors in the code (this approach is sometimes called *bug hunting*).

In this paper, testing and dynamic analysis of concurrent programs are combined with metaheuristic techniques. *Stochastic optimisation techniques*, also called *metaheuristics* or *search-based optimisation* [14], employ a certain degree of randomness in the process of finding as optimal as possible solutions to complex well-defined problems. Such problems commonly have a large space of possible solutions (also known as *search space*) and no known efficient and complete solution. Instead, heuristics are used to partially explore the search space and favour promising parts of the space with good solutions. In order to be able to distinguish suitability of each solution, metaheuristic techniques define the so-called *fitness function* which is problem specific and express the quality of each candidate solution with respect to the chosen goal. With a metaheuristics approach, there is no guarantee to find globally optimal solutions. However, metaheuristics deliver satisfactory solutions for complex problems in a reasonable time.

One of the most popular techniques which is also used in our approach are the *genetic algorithms (GAs)* [14]. GAs are inspired by the evolution processes in nature, handle a set of solutions (called *population*) in memory, and during each iteration determine the next population using

*Recommended by thesis supervisor: Prof. Tomáš Vojnar Defended at Faculty of Information Technology, Brno University of Technology on September 14, 2012.

Letko, Z. Analysis and Testing of Concurrent Programss. Information Sciences and Technologies Bulletin of the ACM Slovakia, Vol. 5, No. 3 (2013) 1-7

a stochastic algorithm containing three major steps called *selection*, *crossover*, and *mutation*. A solution in the genetic algorithm is encoded as a vector of values (usually but not necessarily of boolean values) called a *genome*. The selection operation decides which available solutions (also called *parents*) will be used to infer a new member of the next population. The crossover operation combines two parents, and the mutation operation modifies (mutates) the result of the crossover. GA appropriately combines exploration and exploitation and often can find a good solution for the given problem.

2. Advanced Techniques of Testing Concurrent Programs

A sequential program is executed as a single process with a single thread of control. When it is executed with the same input, the sequential program goes through the same sequence of instructions and provides the same output. This mostly deterministic behaviour makes analysis and verification of sequential programs simpler. On the other hand, a concurrent program is executed within one or more processes including multiple threads of control. Non-deterministic scheduling causes that the multithreaded program executed with the same input multiple times may produce different outputs.

A crucial issue when testing concurrent software is to test as many different (and hopefully relevant) interleavings as possible. To achieve that, one can use *stress testing* which uses more active threads than the number of available processors so that at any given time only some threads are running, thus reducing the predictability of interactions between threads. This technique is however effective only a little. Advanced approaches to test concurrent software, which are described below, are based on a repeated execution of the same test with the same inputs and on detecting whether an error occurred during the execution. Such detection can be based on a failure detection, assertion checking, or some dynamic detection technique. During each execution, the techniques try to affect the scheduler with an intention to see interleavings which have not been spot during the previous executions of the test. The number of different interleavings is increased either by injecting of the so-called *noise* into test executions or by enforcing deterministic scheduling.

Noise injection techniques [4] inject either randomly or based on some heuristics a noise into the test execution. The noise causes a delay in the execution of a selected thread, giving other threads which are ready to run an opportunity to make a progress. An advantage of the noise injection approach is that the method does not require any modification of the execution environment nor a manual modification of the test. The tested system is automatically *instrumented*. The instrumentation typically injects calls to a *noise maker* routine into the program code. Threads executing the modified code then enter the noise maker routine that decides—either randomly or based on some heuristics—whether to cause a noise. Notice that already the instrumentation itself introduce some noise into the execution because the thread must execute the code injected by the instrumentation. The technique is mature enough to be used for testing of real-life software. Interleavings obtained by the noise injection technique are all valid. Therefore, the technique *does not introduce any false alarms*.

Deterministic testing [2] controls thread scheduling decisions during the test execution and systematically explores the interleaving space. Such tools are inspired by the work on model checking and can be seen as lightweight model checking (or execution-based model checking). The tools explore alternative scheduling scenarios using re-initialisation of the tested program. To avoid undesirable modification of the execution environment, modern tools for deterministic testing, e.g., [12], focus on application programming interfaces providing synchronisation functionality to the tested programs. The calls of synchronisation routines of the run-time environment or OS are intercepted and passed to the *deterministic scheduler*. This scheduler is able to stop threads which should not proceed. The system scheduler therefore schedules only threads allowed to run by the deterministic scheduler. The deterministic scheduler resumes stopped threads when needed. Despite the introduced techniques are able to handle large programs thanks to various optimisations, they still suffer from certain limitations. First, they are sensitive to other sources of non-determinism (e.g., input/output events) which make it difficult to replay an already captured scenario. Second, during the replay phase, they usually allow to run only one thread which has a large impact on the performance of the tested program.

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

3. Concurrency Coverage Metrics

In testing, testers need measures that can be used to asses how well a program has been tested, how good a test is, or whether further testing is necessary. For this purpose, the concept of *coverage metrics* is used. Coverage metrics are based on *coverage tasks* representing different phenomena whose occurrence in the behaviour of a tested program is considered to be of interest. Probably the most popular measure is the *code coverage* which measures how much of the code (the number of lines, the number of executed statements, the number of branch conditions covered both ways, etc.) has been executed during a test execution. A high code coverage is a necessary condition for a good verification. The *concurrency coverage* metrics discussed in this chapter measure how well the synchronisation mechanisms and various other concurrency-related aspects of the behaviour has been exercised.

A common goal of the testing process is to reach a *full coverage*, i.e., to cover all tasks of the coverage domain. However, obtaining a full coverage for a complex software and nontrivial metrics is often difficult and expensive. Moreover, for many nontrivial metrics, it is very difficult and in general undecidable to statically determine reachable coverage tasks and hence full coverage. Coverage metrics without a known full coverage can, however, still be used in various ways. First, they can be used for compar-

isons of testing techniques and tests. Second, they can be used to control termination of the testing process within the so-called *saturation-based testing* where the so-called *saturation effect*, i.e., a situation when the obtained coverage stops growing, can be used to determine whether the testing can be stopped. Finally, they are also useful in *search-based testing* discussed in Section 5.

In [9], several new coverage metrics suitable for saturation-based or search-based testing of concurrent programs are provided. These metrics are based on coverage tasks derived from the information about program behaviour that is gathered or computed by various dynamic analyses—Eraser [13], GoldiLocks [5], AVIO [11], and GoodLock [1]. In fact, the idea of inferring new metrics from these analyses is rather generic and can be applied to other dynamic as well as static analyses (even those that will appear in the future) too. The proposal is motivated by the idea that within the development of such analyses, behavioural aspects of concurrent programs that are highly relevant for the existence of synchronisation-related errors have been identified. Hence, it makes sense to measure how well the aspects of the behaviour tracked by such analyses have been covered during testing.

3.1 Methodology of Deriving New Coverage Metrics

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

The described idea is very generic, and one can speak about a new class of concurrency coverage metrics that can be obtained in the described manner. A crucial step in the creation of a new coverage metrics based on some error detection algorithm is to choose suitable pieces of information available to or computed by the detection algorithm, which are then used to construct the domain of the new coverage metrics such that the other, above mentioned criteria are met. This leads to a trade off among the precision of the metrics and the amount of information tracked, the associated computational complexity, and speed of saturation.

4. Noise Injection Heuristics

In this section, heuristics for noise injection are discussed. Furthermore, the results of a *systematic comparison* [10] of several noise injection techniques available in the IBM Concurrency Testing Tool (ConTest), which represents the state of the art of noise injection, as well as our newly proposed coverage-based heuristics [10] on a set of test cases are summarised.

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

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

There also exist multiple noise placement techniques for determining where to put a noise. The problem of noise placement can be divided into two subproblems: (i) Which program locations are suitable for injection of noise and (ii) which particular occurrence of selected program locations in an execution of the program actually affect by the noise. IBM ConTest allows to inject a noise before and after any concurrency-related event (including, accesses to class member variables, static variables, and arrays, and the calls of `wait`, `interrupt`, `notify`, `monitorenter`, and `monitorexit` routines).

4.1 Suggestions for Noise-based Testing

The previously published systematic comparison of noise injection heuristics [10] shows that there is no silver bullet among the many existing noise seeding and noise placement heuristics. Moreover, it identifies weak and strong aspects of the different heuristics in different contexts and can thus serve as a guide for a user which intend to apply the heuristics in the testing process. Apart form that, the comparison also shows that the newly proposed heuristics may in certain cases provide an improvement in the testing process. The results of the comparisons are summarised within several suggestions for noise-based testing presented below.

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

Otherwise, one has to set up the noise seeding and placement heuristics manually. As for noise seeding heuristics, good results were often provided by the *yield*, *synchYield*, *wait*, and *busyWait* heuristics. The *yield* and *synchYield* heuristics have a minimal impact on the performance of the test while still providing the best improvement in some cases. The *wait* and *busyWait* heuristics cause a considerable performance degradation but they can help to

test even rarely executed synchronisation scenarios in complex programs. The results indicate that in most cases higher noise frequency does not mean a higher probability of spotting an error or higher coverage. On the other hand, a high noise frequency used with a demanding heuristics (e.g., *busyWait*) has a negative impact on the performance of the test.

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

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

To sum up, although we provided same hints on using the noise techniques above, these advises are not definite since different testing scenarios can quite significantly vary as we proved by our experiments. Hence, if it is possible, we suggest to experiment with more noise settings.

5. Search-based Testing of Concurrent Programs

Search-based testing applies metaheuristic search techniques [14] to the problem of software testing. In order to apply metaheuristics to software engineering problems like testing, one has to consider the following steps [3]: (i) Decide whether the problem is suitable for search-based techniques, (ii) formulate the problem as a search/ optimisation problem and define a representation for the possible solutions, (iii) define the fitness function, (iv) start with the simple Hill-climbing algorithm—if the results are encouraging, i.e., better than random search, consider other local search and genetic approaches, and (v) select an appropriate metaheuristic technique, its parameters and operators if necessary.

Above, we show that there is no silver bullet among the many existing noise injection heuristics. Actually, some configurations can even decrease the probability of an error manifestation. This is helpful for run-time healing of errors [7], but it is highly undesirable for detecting them. Moreover, the number of possible settings of the noise injection (and also of the test itself) together with the considerable time needed to run a test in order to evaluate the efficiency of a certain noise configuration makes exhaustive searching for suitable noise configurations impractical. This is exactly the case where metaheuristic search techniques can help.

5.1 Concurrent Programs Testing as a Search Problem

Most types of the noise seeding or placement 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. Finally, it is usually the case that there exist multiple test cases for a given program that can also be parametric.

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

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

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

Metaheuristic algorithms need an objective function in order to compute suitability of the candidate solution (the fitness function). Fitness of a test configuration $tc \in TC$ wrt. the 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. 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 [8], our initial experiments done with the basic local search algorithm—the Hill-climbing algorithm [14] are presented. In the experiment, the Hill-climbing algorithm is compared with the random search approach for solving the simple TNCS problem. Our results indicate that the metaheuristic algorithms can be used to solve the TNCS problem because even the simple Hill-climbing algorithm is in some cases able to overcome the random approach. But, the Hill-climbing algorithm often gets stuck in a local optimum. The landscape analysis indicates that the landscape contains many local optima which are hard to overcome for the simple Hill-climbing algorithm.

5.2 A Genetic Approach to the TNCS Problem

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.

We consider the standard one-point, two-point, and uniform element-wise (any-point) *crossover operators* [14]. *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 [14] (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 [14] as they are implemented in the ECJ library.

Based on the results presented in [6], we found as suitable the following setting of the parameters of genetic algorithms for the considered concretisation of the TNCS problem: Size of population 20, two different selection operators (tournament among 4 individuals and fitness proportional), the *any-point* crossover with a higher probability (0.25), a low mutation probability (0.01), and two elites (that is 10 % of the population). We choose the low mutation probability 0.01 despite our results indicate that the individuals with highest fitness are most often found using the higher probability (0.25). This decision is motivated by our intention to prefer exploitation over exploration as explained below. This parameter setting is used in the experiments presented below.

In [6], we proposed a complex objective function for the TNCS problem. 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. Namely, we aim at (1) maximising weighted coverage under the concurrency coverage metric *GoldiLockSC* [9] (denoted as $WGoldiLockSC$ below), (2) maximising the number of warnings $GLwarn$ produced by the GoldiLocks algorithm [5] (denoted as $GLwarn$), (3) maximising the number of detected real errors due to data races (denoted as $error$), and (4) minimising the execution $time$. Our fitnees function has form:

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

6. Experimental Results

We evaluate our approach on 5 test cases containing concurrency-related errors. The test cases are listed in Table 1. In the table, the *Param* column indicates the number of the test case parameters and the number of possible values of each parameter (e.g., $2, 3$ means that the test takes two parameters, the first with two possible values and the second with three possible values).

We compare our genetic approach with the random approach which represents the state of the art in the noise-based testing of concurrent programs. 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.

6.1 An Evaluation of the Best Individuals

The middle part of Table 1 (*Best configuration*) contains three columns which compare the best individual obtained by our genetic approach and found by the random approach. The *Gen.* column contains the average number of generations (denoted as *gen* below) within which we discovered the best individual according to the considered fitness function. The numbers indicate that we are able to find the best individual according to the considered fitness function within the first quarter of the considered generations.

The *Error* column of the *Best configuration* 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 individual found randomly in 2000 evaluations. The –/– value represents a situation where none of the best individuals was able to detect the error within the allowed 5 executions. The ★ 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

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

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

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

6.2 An Evaluation of the Search Process

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

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

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

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

7. Conclusions

In this paper, we have concentrated on noise injection techniques that help to examine different thread interleavings during testing and dynamic analysis of concurrent programs and to detect even rarely manifesting errors. Our main contribution can be divided into three parts concerning concurrency coverage metrics, noise injection heuristics, and a use of metaheuristics in noise-based testing.

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

In the next part of the paper, we have provided a summary of comparison of different *noise injection heuristics* including the new heuristics which we proposed in [10]. Our results have shown that there is no silver bullet among the noise seeding and placement heuristics. Even our newly proposed heuristics wins over the existing ones only in some cases.

Finally, we have proposed a way of using search techniques to improve quality of noise-based testing and dynamic analysis through finding suitable combinations of parameters of tests and noise heuristics. We have formalised this problem as the *test and noise configuration search (TNCS) problem.* We have proposed a way how to use genetic algorithms to solve the TNCS problem and a complex objective function suitable for data race detection which is based on a dynamic analysis algorithm (namely, the GoldiLocks algorithm). Our experiments has shown that the objective function has been also successful at looking for other kinds of concurrency errors. We have shown on a set of benchmark programs that our approach significantly outperforms the commonly used approach of randomly selecting noise configurations.

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-S-11-1 and FIT-S-12-1.

References

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

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

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

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

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

[6] V. Hrubá, B. Křena, Z. Letko, and T. Vojnar. Testing of concurrent programs using genetic algorithms. Accepted for publication at SSBSE'12, 2012.

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

[8] B. Křena, Z. Letko, R. Tzoref, S. Ur, and T. Vojnar. A Platform for Search-based Testing of Concurrent Software. In *PADTAD'10*, pages 48–58, New York, NY, USA, 2010. ACM.

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

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

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

[12] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and Reproducing Heisenbugs in Concurrent Programs. In *Proc. of Symposium on Operating Systems Design and Implementation*—OSDI'08, pages 267–280, Berkeley, CA, USA, 2008. USENIX Association.

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

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