# Noise Injection Heuristics for Concurrency Testing

Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar

FIT, Brno University of Technology, Czech Republic

**Abstract.** Testing of concurrent software is difficult due to the non-determinism present in scheduling of concurrent threads. Existing testing approaches tackle this problem either using a modified scheduler which allows to systematically explore possible scheduling alternatives or using random or heuristic noise injection which allows to observe different scheduling scenarios. In this paper, we experimentally compare several existing noise injection heuristics both from the point of view of coverage of possible behaviours as well as from the point of view of error discovery probability. Moreover, we also propose a new noise injection heuristics which uses concurrency coverage information to decide where to put noise and show that it can outperform the existing approaches in certain cases.

## 1 Introduction

Concurrency software testing and analysis is hard due to the non-deterministic nature of scheduling of concurrent threads. Static analysis and model checking do not scale well when analysing such programs due to the large interleaving space they need to explore. Testing and dynamic analysis scale well but usually do not analyse all possible interleavings. The number of different interleavings spot during repeated executions of the same test can be increased either by using a *deterministic scheduler* or by injecting of so-called *noise* into test executions.

Deterministic schedulers [12] control thread scheduling decisions during a program execution and so can systematically explore the interleaving space up to a certain extent. Such tools can be seen as light-weight model checkers. Noise injection tools [3, 11] inject 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. The noise causes a delay in the current thread, giving other threads opportunity to make a progress.

Coverage metrics are used to measure how many *coverage tasks* (i.e., monitored events such as reachability of a certain line) defined by a *coverage model* have been covered during test execution(s) so far. Concurrency coverage metrics [14, 2, 7] can be used to track how many different concurrency-related tasks have been covered, and hence how many different interleavings have been witnessed.

This paper presents two contributions to the research on noise injection techniques. First, we propose a *new heuristics* which uses coverage information to

select places in an execution of a given code where to put a noise. We also propose a way to determine the strength of the noise needed to suitably affect the behaviour of tested programs. Second, we address the current lack of experimental evaluations of the various noise injection heuristics by *systematic comparison* of several noise injection techniques available in the well-known IBM Concurrency Testing Tool (ConTest) [3] as well as our new heuristics on a set of test cases of different size. The comparison is based on the coverage obtained under one selected concurrency coverage metric, the needed execution time, and the rate of manifestation of concurrency errors in the testing runs.

We in particular focus on concurrent programs written in Java. We use our infrastructure for search-based testing called SearchBestie [8] to run multiple tests with different parameters and to collect their results and IBM ConTest [3] for noise injection and concurrency coverage measurement. Although our comparison could certainly be further extended, we believe that the comparison provides results missing in the existing literature on noise injection. Moreover, the comparison shows that our new heuristics may in certain cases provide an improvement in the testing process.

## 2 Existing Noise Injection Heuristics

Existing works discuss three main aspects of heuristic noise injection: (1) how to make noise, i.e., which type of noise generating mechanism should be used, (2) where to inject noise during a test execution, i.e., at which program location and at which of its executions, and (3) how to minimise the amount of noise needed for manifestation of an already detected error when debugging. This work mainly targets the first two aspects. More information on debugging can be found, e.g., in [5].

There exist several ways how a scheduler decision can be affected in Java. In [3], three different noise seeding techniques are introduced and evaluated on a single-core processor. The *priority* technique changes priorities of threads. This technique did not provide good results. The *yield* technique injects one or more calls of `yield()` which causes a context switch. The *sleep* technique injects one call of `sleep()`. Experiments showed that the sleep technique provided best results in all cases. However, when many threads were running, the yield technique was also effective.

The current version of the IBM ConTest tool comes with several more noise seeding techniques [9]. The *wait* technique injects a call of `wait()`. The concerned threads must first obtain a special shared monitor, then call `wait()`, and finally release the monitor. The *synchYield* technique combines the yield technique with obtaining the monitor as in the wait technique. The *busyWait* technique does not obtain a monitor but instead loops for some time. The *haltOneThread* technique [13] occasionally stops one thread until any other thread cannot run. Finally, the *timeoutTamper* heuristics randomly reduces the timeout used when calling `sleep()` in the tested program (to test that it is not used for synchronisation). All the above mentioned seeding techniques except the priority technique are parameterised by the so-called *strength of noise*. In the case

of techniques based on `sleep` and `wait`, the strength gives the time to wait. In the case of `yield`, the strength says how many times the yield should be called.

Next, we discuss techniques for determining where to put a noise. IBM ConTest allows to inject a noise before and after any concurrency-related event (namely, access to class member variables, static variables, and arrays, calls of `wait`, `interrupt`, `notify`, `monitorenter`, and `monitorexit` routines). The *rstest* [11] tool considers as possibly interesting places before concurrency-related events only. Moreover, rstest uses a simple escape analysis and a lockset-based algorithm to identify so-called *unprotected accesses* to shared variables. The unprotected access reads or writes a variable which is visible to multiple threads without holding an appropriate lock. This optimisation reduces the number of places where the noise can be put but suppresses ability to detect some concurrency errors, e.g., high-level data races or deadlocks where all accesses to problematic variables are correctly guarded by a lock.

It is discussed in [3, 11, 5] that putting noise on every possible place is inefficient and only a few relevant context switches are critical for the concurrency error. Also, putting noise in a certain place (*ploc*—program location [3]) in the execution can either help to spot the concurrency error or mask it completely. Therefore, several heuristics for choosing places where to put a noise were proposed, e.g., in [3, 11, 1, 6, 4, 13].

The simplest heuristics is based on a *random noise* [3, 11]. This heuristics puts a noise before/after an executed *ploc* with a given probability. The probability is the same for all *ploc*s in the execution. It was shown in [1] that focusing random noise only on a single variable over which a data race exists increases the probability of spotting the error. The authors also propose a heuristics which helps to choose a suitable variable without additional information from a data race detector. In [4], the noise injection problem is reformulated as a *search problem*, and a genetic algorithm is used to determine *ploc*s suitable for noise injection. The fitness function used prefers solutions with a low number of *ploc*s where a noise is put (size), a high amount of noise in less *ploc*s (entropy), and a high probability of spotting the error (efficiency). In [6], several *concurrency antipatterns* are discussed, and for each of them, a suitable scheduling scenario that leads to manifestation of the corresponding concurrency error is presented, but the paper contains no practical evaluation of the proposed heuristics.

A few heuristics based on concurrency coverage models have been published. Coverage-directed generation of interleavings presented in [3] considers two coverage models. The first model determines whether the execution of each method was interrupted by a context switch. The second model determines whether a method execution was interrupted by any other method. The level of methods used here is, according to our opinion, too coarse. In [13], a coverage model considers, for each synchronisation primitive, various distinctive situations that can occur when the primitive is executed (e.g., in the case of a synchronised block defined using the Java keyword `synchronised`, the tasks are: *synchronisation visited*, *synchronisation blocking* some other thread, and *synchronisation blocked* by some other thread). A forcing algorithm then injects noise at correspond-

ing synchronisation primitive *ploc*s to increase the coverage. None of these two heuristics focuses on accesses to shared variables which can limit their ability to discover some concurrency errors, e.g., data races.

## 3 A New Coverage-based Noise Injection Heuristics

Our new heuristics is motivated by our recent experiences with concurrency coverage metrics [7]. The heuristics primarily answers the question where to inject noise during a test run (the noise can be caused by any of the `wait`, `sleep`, or `yield` seeding techniques). In the heuristics, we consider only *ploc*s that appear before concurrency-related events as suitable for noise injection. Our heuristics targets both accesses to shared variables as well as the use of synchronisation primitives. Our goal is to be able to discover all kinds of concurrency errors. Our heuristics monitors the frequency of a *ploc* execution during a test and puts a noise at the given *ploc* with a probability biased wrt. this frequency—the more often a *ploc* is executed the lower probability is used. Furthermore, our heuristics also derives the strength of a noise to be used from the timing of events observed in previous executions of the test (although for determining the strength of noise, alternative approaches can be used too).

The testing process with our noise injection heuristics works in the following four steps. (1) No noise is produced, and a set of covered tasks of our coverage metric together with information on relative timing of appearance of monitored concurrency-related events are generated during the first execution of the test. (2) A set of the so-called *noise tuples* is generated from the gathered information. (3) Random noise at the *ploc*s included in the noise tuples is generated, and the average frequency of execution of these *ploc*s within particular threads is gathered during the next test execution. (4) Biased random noise of strength computed wrt. the collected statistics is (repeatedly) produced at the collected *ploc*s. Coverage information is updated during each execution, and new noise tuples are constantly learnt. Likewise, all other collected statistics are updated during each test run. Due to performance reasons, only one thread is influenced by noise at a time. We now explain the above introduced steps in more detail.

Our coverage model considers coverage tasks of the form $(t_1, ploc_1, t_2, ploc_2)$. There are two situations when a task is covered. First, a task is covered if a thread $t_1$ accesses a shared variable $v$ at $ploc_1$, and subsequently a thread $t_2$ accesses $v$ at $ploc_2$, which is a typical scenario critical for occurrence of concurrency-related errors. If $t_1$ owns a monitor when accessing $v$ at $ploc_1$, another task $(t_1, ploc_3, t_2, ploc_2)$ where $ploc_3$ refers to the location where $t_1$ obtained the last monitor is also covered. This is motivated by considering the relative position of locking a critical section in one thread and using it in another thread as important. Second, a new task is covered if a thread $t_1$ releases a monitor obtained at $ploc_1$, and subsequently a thread $t_2$ obtains the monitor at $ploc_2$. Each covered task is annotated by the number of milliseconds that elapsed between the events on which the task is based. The threads are identified in an abstract way based on the history of their creation in the same way as in [7].

Our heuristics injects noise before a location $ploc_1$ executed by a thread $t_1$ if a task $(t_1, ploc_1, t_2, ploc_2)$ has been covered within some previous execution. This way, our heuristics tries to reverse the order in which the locations are executed. The coverage information collected during previous runs is transformed into *noise tuples* of the form $(t_1, ploc_1, min, max, orig, exec)$. Here, $t_1$ identifies a thread and $ploc_1$ the program location where to put a noise. The two next values give the minimal and maximal number of milliseconds that elapsed between the events defining the given coverage task. These values can be used for determining the strength of noise to be used as a delay of length randomly chosen from between the values. If there are multiple coverage tasks with the same couple $(t_1, ploc_1)$, $min$ and $max$ are computed from all such tasks. The $orig$ value contains an identification of the run where the couple $(t_1, ploc_1)$ was spot for the first time. In order to limit values of $min$ and $max$, their update is possible only within a limited number of test executions after the $orig$ run. Finally, the $exec$ value contains the average number of times the couple $(t_1, ploc_1)$ is executed during a test execution. It is used to bias the probability of noise injection at $ploc_1$.

In repeated executions of a test, the so far computed noise tuples are loaded, and the noise is generated at program locations given by them with the probability computed from the number of times the locations have been executed (the $exec$ value). The computation is shown in Alg. 1. The base probability is obtained as $1/exec$ to be higher for $ploc$s that are executed rarely. The minimal noise probability accepted by ConTest is 0.001, and so all lower computed probabilities are set to this value. Higher probabilities are divided by 4 to keep the noise injection frequency reasonably low (25 % for a $ploc$ which is executed once during each test). This is motivated by our observation that higher probability than 25 % degrades test performance and usually does not provide considerably better results. The limit can be changed if necessary. If the $exec$ value is not yet available, the probability of 0.01 is used.

```
1 if exec > 0 then
2     prob = 1 / exec;
3     if prob < 0.004 then
4         prob = 0.001;
5     else prob = prob/4;
6 else  prob = 0.01;
```

**Alg. 1.** Computing probability of noise generation

## 4 A Comparison of Noise Injection Techniques

This section presents an experimental evaluation of selected noise injection techniques available in ConTest as well as of the above newly proposed heuristics. We evaluate these techniques on a set of 5 test cases shown in Table 1 which gives the number of classes the test cases consist of and the concurrency error present in them (if there is one). The *sunbank* test case runs 4 threads representing bank clients each performing a set of transfers. There is a data race on a variable containing the total amount of money in the bank. The *airlines* [7] test case represents an artificial air ticket reservation system. During each test, 4 threads representing ticket resellers serve requests of 8 client threads. The test case contains a high-level atomicity violation.

The three other programs in Table 1 are real-life case studies. The *crawler* case study [1] is a skeleton of an older version of a major IBM software. In this test case, 16 threads simulate serving of remote requests. A data race can manifest here if a certain very rare timing condition is met during a shutdown sequence. The *ftpserver* case study [7] is an early development version of an open-source FTP server. The server creates a new thread for each connection. The code contains several data races, out of which we focus only on those producing a `NullPointerException`. Finally, the *tidorbj* test case is an open source CORBA-compliant object resource broker [10]. We used the *echo_concurrent* test case available in the distribution. The test starts 10 clients, each sending a set of requests to the server. This test case does not contain any known concurrency error.

**Table 1.** Test cases

| Test | Classes | Concur. error |
|---|---|---|
| sunbank | 2 | data race |
| airlines | 8 | atom. viol. |
| crawler | 19 | data race |
| ftpserver | 120 | data race |
| tidorbj | 1399 | none |

During each test run, we measure coverage wrt. a chosen metric—namely, *Avio** [7]. This metric has been chosen due to its very good ratio of providing good results from the point of view of suitability for saturation-based or search-based testing and a low overhead of measuring the achieved coverage (and hence its suitability for performing many tests with minimal interference with tested programs—still, in the future, more experiments with other metrics could be done). Note that the Avio* metric that we use for evaluation of the testing is different than the specialised metric that we have proposed above as a means for driving the noise injection. In particular, the *Avio** coverage metric focuses on accesses to shared variables and collects triplets consisting of two subsequent accesses $a1, a3$ to a shared variable $v$ from a thread and the last access $a2$ to $v$ from another thread that interleaved accesses $a1$ and $a3$. Besides coverage information, we monitor execution times and occurrences of the known errors. Collection of this information of course affects thread scheduling of the monitored test cases, but the influence is the same for all performed executions. All tests were executed on multi-core machines running Linux and Java version 1.6.

To recall from Section 2, IBM ConTest provides 5 basic techniques for noise seeding: *yield*, *sleep*, *wait*, *busyWait*, and *synchYield*. In addition, the so-called *mixed* technique simply randomly chooses one technique from the others. The probability of causing a noise at a *ploc* is driven by the *noise frequency* (*nFreq*) parameter ranging from 0 (no noise) to 1000 (always). We limit this parameter to values 0, 50, 100, 150, and 200. Higher values cause significant performance degradation and are therefore not considered. The mentioned basic noise seeding techniques can be combined in ConTest with two further techniques—*haltOneThread* and *timeoutTamper*. The approach of setting a certain noise frequency to control when some noise is generated can then be combined with restricting the noise generation to events related to (certain) shared variables (the *sharedVar* heuristics). Finally, ConTest provides a so-called *random* setting under which it randomly selects and combines its parameters.

Each of our 5 test cases was tested with 496 different noise injection configurations. We collected data from 60 executions for each configuration. This way,

**Table 2.** Relative improvement of error detection when using different types of noise

| test case | nFreq | sleep | busyWait | wait | sYield | yield | mixed | average |
|---|---|---|---|---|---|---|---|---|
| sunbank | 50 | 1.63 | 1.32 | 2.28 | 0.45 | 1.60 | 0.85 | 1.36 |
| | 100 | 3.05 | 2.48 | 4.22 | 0.00 | 0.38 | 3.62 | 2.29 |
| | 150 | 4.18 | 1.68 | 2.52 | 0.00 | 4.85 | 2.03 | 2.54 |
| | 200 | 3.85 | 3.12 | 6.13 | 0.00 | 4.47 | 2.50 | 3.34 |
| airlines | 50 | 1.13 | 1.13 | 0.65 | 0.67 | 3.06 | 0.91 | 1.26 |
| | 100 | 2.44 | 1.45 | 1.34 | 1.88 | 5.48 | 1.35 | 2.32 |
| | 150 | 0.21 | 1.89 | 1.42 | 1.83 | 5.21 | 0.47 | 1.84 |
| | 200 | 1.90 | 0.23 | 0.58 | 1.93 | 5.54 | 1.15 | 1.89 |
| ftpserver | 50 | 0.36 | 0.34 | 0.56 | 0.94 | 0.91 | 0.49 | 0.60 |
| | 100 | 0.21 | 0.48 | 0.28 | 0.90 | 0.96 | 0.35 | 0.53 |
| | 150 | 0.36 | 0.22 | 0.30 | 0.98 | 0.95 | 0.31 | 0.52 |
| | 200 | 0.20 | 0.60 | 0.29 | 0.99 | 0.90 | 0.31 | 0.55 |
| average | | 1.63 | 1.24 | 1.71 | 0.88 | 2.86 | 1.19 | 1.59 |

we obtained a database of 148,800 results. Then, we computed average cumulated values for sequences of 1, 10, 20, 30, 40, and 50 randomly chosen results of each configuration (the length of the sequence is denoted as *SeqLen* below). These average results represent average values that one obtains when executing the given configuration *SeqLen* times.

Due to limited space, only two analyses of the results are presented here: (1) A comparison of the efficiency of the different noise seeding techniques available in ConTest together with the influence of the noise frequency on them. (2) A comparison of the efficiency of the ConTest's heuristics restricting noise generation to events related to (certain) shared variables and our newly proposed heuristics for deciding where to generate noise in a testing run. Hence, the first comparison is mainly about the types of noise seeding and partially about where the noise is generated in a test execution whereas the second comparison is mainly about the latter issue.

### 4.1 A Comparison of ConTest's Noise Seeding Settings

In this subsection, we focus on the influence of the different noise seeding techniques and the noise frequency on how the testing results are improved in comparison to testing without noise injection (but with the collection of data about the testing enabled, which also influences the scheduling). Since ConTest does not allow one to use its *timeoutTamper* and *haltOneThread* noise seeding techniques without one of its basic noise seeding techniques, we first study the effect of the basic noise seeding techniques, which are activated via the `noiseType` parameter of ConTest. Then we focus on the effect of the *timeoutTamper* and *haltOneThread* seeding techniques.

Table 2 shows the relative improvement of error detection that we observed when using different basic noise seeding techniques available in ConTest. Both the *haltOneThread* and *timeoutTamper* seeding techniques were disabled, the

random noise injection heuristics was enabled, and *SeqLen=50*. Additionally, we also consider the ConTest setting which randomly chooses among basic noise seeding techniques before each test execution (referred as *mixed* in the table). The entries of the table give the ratio of the number of error manifestations observed when using noise injection of the respective type against the number of error manifestations without any noise setting enabled. Moreover, average values are provided for a better comparison. Values lower than 1.00 mean that the appropriate configuration provided a worse result than without noise. Higher values mean that noise of the appropriate type provides better results. For instance, 1.25 means that the given type of noise on average detected an error by 25 % more often. Results for the *crawler* and *tidorbj* test cases are omitted because there were no errors detected by the considered test configurations in those test cases.

The table illustrates that noise injection affects each test case differently—sometimes it helps, sometimes not. The use of noise almost always very significantly helps in the cases of *sunbank* and *airlines*, but it does not help in the case of *ftpserver*. Also, the different seeding techniques perform differently in the different test cases, and one cannot claim a clear winner among them (although *yield* seems to be often winning). The *wait* technique helps the most in the *sunbank* test case while *yield* provides the best improvement in the *airlines* test case. In the case of *ftpserver*, no technique provides improvement. Significant influence of *nFreq* is visible in the *sunbank* test case, but in the *ftpserver* case, it seems that *nFreq* has no influence. The effect of *nFreq* in *airlines* has no clear tendency. Nevertheless, overall, the table demonstrates that choosing a suitable noise seeding technique can rapidly improve the probability of detecting an error at least in some cases.

Further, we have also performed experiments on how using the different basic noise seeding techniques available in ConTest impacts upon coverage obtained under the Avio* metric. The obtained results can be summarised by saying that the obtained improvement due to the use of noise injection was smaller in this case, and the differences among the noise seeding techniques were smaller too. The best improvement was achieved using the *busyWait* technique (about 45 %) in the *crawler* and *ftpserver* test cases.

Table 3 shows influence of the *timeoutTamper* and *haltOneThread* noise seeding techniques as well as their combination on error detection (in the table, t_0/t_1 indicates whether *timeoutTamper* is disabled or enabled, and h_0/h_1 indicates whether *haltOneThread* is disabled or enabled, respectively). As said above, these techniques cannot be used without any basic noise seeding techniques in ConTest, and therefore average values computed from results obtained with different basic noise seeding techniques are reported. Results for *nFreq=150* and *seqLen=50* are used. Like in Table 2, the ratio of the number of manifested errors against the number of manifested errors when no noise is used is presented.

**Table 3.** Influence of the *haltOneThread* and *timeoutTamper* techniques on error detection

| | t_0 | | t_1 | |
|---|---|---|---|---|
| test case | h_0 | h_1 | h_0 | h_1 |
| sunbank | 2.54 | 0.95 | 1.93 | 1.72 |
| airlines | 1.84 | 2.55 | 1.61 | 2.29 |
| ftpserver | 0.52 | 0.61 | 0.34 | 0.45 |

The table shows that *timeoutTamper* and *haltOneThread* also affect each test case differently. The *haltOneThread* technique significantly helps in the *airlines* test case, slightly helps in the *ftpserver*, but it is harmful in the *sunbank* test case. The *timeoutTamper* technique provides worse results in all shown test cases. On the other hand, in the *crawler* test case (not shown in the table since no error is detected in it without noise injection), testing with *timeoutTamper* enabled and *haltOneThread* disabled was the only configuration of ConTest that allowed an error to manifest (in 7 % of the executions).

Table 4 illustrates the influence of the *timeoutTamper* and *haltOneThread* noise seeding techniques on the coverage obtained under the Avio* coverage metric. The table clearly shows that the effect of *timeoutTamper* is very important for the *crawler* test case. As we have already said, this test case is a skeleton of one IBM software product. When developers extracted the skeleton, they modeled its environment using timed routines. The *timeoutTamper* heuristics influences these timeouts in a way leading to significantly better results. The effects of the considered techniques in the other examples are then none or very small.

**Table 4.** Influence of the *haltOneThread* and *timeoutTamper* techniques on Avio* coverage

| | t_0 | | t_1 | |
|---|---|---|---|---|
| test case | h_0 | h_1 | h_0 | h_1 |
| sunbank | 1.04 | 1.03 | 1.04 | 1.06 |
| airlines | 0.85 | 0.81 | 0.77 | 0.86 |
| crawler | 1.15 | 1.30 | 3.91 | 3.78 |
| ftpserver | 1.04 | 1.04 | 1.09 | 1.07 |
| tidorbj | 0.95 | 0.95 | 0.95 | 0.96 |

The same trends as described above can also be seen from results of experiments that we have performed with different values of *nFreq*. Our results indicate that there is no optimal configuration, and for each test case and each testing goal, one needs to choose a different testing configuration. For instance, the best configuration for the *crawler* test case is a combination of the *busyWait* and *timeoutTamper* noise seeding techniques with *nFreq* set to 200 if the goal is to increase the error detection probability. On the other hand, the testing configuration with *yield*, *timeoutTamper*, and *nFreq* set to 150 provides the best improvement of the Avio* coverage in this test case. In some cases, using a random injection of noise does not provide any improvement as can be seen from the *tidorbj* test case. A significant improvement in this case is achieved only when the noise heuristics discussed in the following section are used.

### 4.2 A Comparison of Heuristics for Determining Where to Generate Noise

This subsection concentrates on the influence of the ConTest's heuristics restricting noise generation to events related to shared variables and on the influence of our new heuristics proposed in Section 3. In addition, the scenario in which ConTest randomly chooses its own parameters is also considered. In particular, Table 5 compares the mentioned heuristics according to the number of Avio* covered tasks divided by the time needed to execute the tests. Intuitively, this relativised comparison favours techniques that provide a high coverage with a low overhead, and therefore punishes techniques that either put too much noise into test executions or provide a poor coverage only. Based on our experiments, we

**Table 5.** A relativised comparison of heuristics restricting places where to put noise

| position | configuration | airlines | crawler | ftpserver | sunbank | tidorbj | average |
|---|---|---|---|---|---|---|---|
| 1 | 0_1_1_1-0-one_0-0 | 7.0 | 2.7 | 9.2 | 4.8 | 7.7 | 6.3 |
| 2 | 0_1_0_1-0-one_0-0 | 5.0 | 2.7 | 11.5 | 4.5 | 8.8 | 6.5 |
| 3 | 0_0_0_0-0-all_1-0 | 2.6 | 17.7 | 2.5 | 7.6 | 2.3 | 6.6 |
| 4 | 1_0_0_0-0-all_0-0 | 10.3 | 3.2 | 10.3 | 5.3 | 5.0 | 6.8 |
| 5 | 0_1_0_1-1-one_0-0 | 11.8 | 5.0 | 11.8 | 4.0 | 8.7 | 8.3 |
| 6 | 0_1_1_1-1-one_0-0 | 9.5 | 7.5 | 15.7 | 3.5 | 7.2 | 8.7 |
| 7 | 0_0_1_1-0-one_0-0 | 5.2 | 18.0 | 3.0 | 9.7 | 9.3 | 9.0 |
| 8 | 0_0_0_1-0-one_0-0 | 6.3 | 17.8 | 2.8 | 10.0 | 8.7 | 9.1 |
| 9 | 0_0_0_1-1-one_0-0 | 8.7 | 15.8 | 10.8 | 7.8 | 7.0 | 10.0 |
| 10 | 0_0_1_1-1-one_0-0 | 10.8 | 14.5 | 9.5 | 10.5 | 7.5 | 10.6 |
| 11 | 0_0_0_0-0-all_0-0 | 5.0 | 19.2 | 11.0 | 12.0 | 10.5 | 11.5 |
| 12 | 0_0_0_0-0-all_1-1 | 3.7 | 23.0 | 19.0 | 6.7 | 13.0 | 13.1 |
| 13 | 0_1_0_1-0-all_0-0 | 14.7 | 5.5 | 19.0 | 15.3 | 14.7 | 13.8 |
| 14 | 0_1_0_0-0-all_0-0 | 17.2 | 7.5 | 17.0 | 12.3 | 16.0 | 14.0 |
| 15 | 0_1_0_1-1-all_0-0 | 17.3 | 6.8 | 18.2 | 11.0 | 17.2 | 14.1 |
| 16 | 0_1_1_1-0-all_0-0 | 17.8 | 6.5 | 13.7 | 17.3 | 15.5 | 14.2 |
| 17 | 0_1_1_1-1-all_0-0 | 14.3 | 9.3 | 16.0 | 16.8 | 14.7 | 14.2 |
| 18 | 0_0_1_1-0-all_0-0 | 13.0 | 14.3 | 9.5 | 19.8 | 16.7 | 14.7 |
| 19 | 0_0_0_1-1-all_0-0 | 14.0 | 16.8 | 14.3 | 16.0 | 13.7 | 15.0 |
| 20 | 0_1_1_0-0-all_0-0 | 16.2 | 10.3 | 17.8 | 14.8 | 16.5 | 15.1 |
| 21 | 0_0_1_1-1-all_0-0 | 15.3 | 16.0 | 9.7 | 19.8 | 15.2 | 15.2 |
| 22 | 0_0_0_1-0-all_0-0 | 18.2 | 17.3 | 11.8 | 16.3 | 17.3 | 16.2 |
| 23 | 0_0_1_0-0-all_0-0 | 19.2 | 18.5 | 9.8 | 19.3 | 18.0 | 17.0 |

have also compared the heuristics according to the number of Avio* covered coverage tasks only (thus providing a non-relativised comparison) as well as according to how often an error is manifested (either taking into account the needed testing time or not). Due to space restrictions, we do not present these latter comparisons in detailed tables here, but we summarize them in the text.

The *configuration* column of Table 5 describes the considered noise injection configuration. A configuration consists of five parts delimited by the "_" character. The meaning of these parts is as follows: (Part 1) The ConTest *random* parameter: if set to 1, ConTest parameters considered in Parts 2–4 are set randomly before each execution. (Part 2) If set to 1, the *timeoutTamper* heuristics is enabled. (Part 3) If set to 1, the *haltOneThread* heuristics is enabled. (Part 4) This part is divided into three sub-parts delimited by "-". The first sub-part indicates whether the ConTest's heuristics limiting noise generation to events related to shared variables is enabled. The second sub-part says whether the noise is also put to other *ploc*s than accesses to shared variables. Finally, the third sub-part says whether the noise is put to *all* shared variables or *one* randomly chosen before each execution. (Part 5) This last part encodes the setting of our noise injection heuristics. It consists of two sub-parts delimited by "-".

The first sub-part says whether our noise injection heuristics is enabled and the second one whether our noise strength computation is enabled too. For further information concerning ConTest configuration, we refer the reader to Section 2 or ConTest documentation [9].

For each considered test case (i.e., airlines, crawler, etc.), we rank the test configurations according to the obtained results—rank 1 is the best, rank 23 is the worst. More precisely, the entries of the table under the particular test cases contain average ranks obtained across the different basic noise types of ConTest. The average rank over all the test cases is provided in the last column. The test configurations are then sorted according to their average rank, giving us their final position in the evaluation of the 23 configurations. We use the final position to identify the configurations in the following text.

As before, the table shows that the efficiency of the different heuristics vary for different test cases. Our heuristics (at position 3) achieved the best results in three out of five test cases (*airlines*, *ftpserver*, and *tidorbj*). The heuristics was not evaluated as the overall winner due to the poor results that it achieved in the *crawler* test case. On the other hand, our heuristics was evaluated as the best for *crawler* when considering the probability of error detection. In fact, there were only three configurations (3, 16, and 18) which were able to detect the very rarely manifesting error in the *crawler* test case. Our heuristics increased the probability of spotting the error the most and achieved the best result in both relativised and non-relativised comparisons. In the other considered test cases, our heuristics was always in the first third of the average results when considering the relativised probability of error detection. As for results of our heuristics in the non-relativised cases of both the Avio* coverage and the probability of error detection, our heuristics achieved worse results (still mostly being in the first half of all the configurations). Hence, based on the results, we can claim that our new heuristics seems to be a good choice when one needs to test bigger programs, especially when having a limited time for testing.

The use of our noise injection heuristics combined with the newly proposed noise strength computation ended at position 12 in Table 5. The results achieved in the various test cases differ more significantly for this configuration than when using the new noise injection heuristics only. Relatively good results were obtained for the *sunbank* and *airlines* test cases, bad results for the other test cases. Similar results were obtained for the relativised Avio* coverage. This is caused by the newly proposed noise strength computation that sometimes puts a considerable amount of noise to places where it might be interesting. This leads to poor results in relativised comparisons where the time plays an important role. On the other hand, the use of our noise injection heuristics combined with the newly proposed noise strength computation provided better results than using our noise injection heuristics only in the non-relativised comparisons because it was able to examine more different interleavings. It was even evaluated as the best for the *tidorbj* test case in the non-relativised comparison using the Avio* coverage. To sum up, we may advice to use the combination of both of the newly proposed heuristics to test bigger programs when performance degradation is not a problem.

Table 5 also clearly shows effectiveness of the ConTest's shared variable heuristics focused on a single randomly chosen shared variable. Configurations based on this heuristics occupy eight from the ten first positions in the table and provide good results also in other considered comparisons. The overall best results were obtained by the combination of this heuristics with the *timeoutTamper* and *haltOneThread* noise heuristics (position 1), which is again mainly due to the effect of the *timeoutTamper* heuristics in the *crawler* test case. Hence, our results prove conclusions presented in [1] that focusing noise on a single variable randomly chosen for each test execution improves the overall test coverage.

Our results then also show that some heuristics trying to restrict the position where to put noise in an intelligent way provide worse results than the configuration with generating noise at random places in test executions (position 11). Finally, we have to admit that surprisingly good results were often provided by the random setting of ConTest too (position 4). This approach provided good results especially in the relativised comparisons and the best result for the *airlines* test case and the criterion of maximizing the probability of error manifestation. Results of this configuration were of course considerably worse for the non-relativised comparison where the execution time is not considered. We suggest to use this configuration when the execution time is important, and one has no idea how the test case is affected by different noise injection techniques.

Results presented in Table 5 were computed for *seqLen=20* and *nFreq=150*. We also examined the influence of changing these paramaters. Our results show that *seqLen* has a minimal impact on the results. Configurations that were evaluated as good after 10 executions of the test were very similarly rated after 50 executions. The *nFreq* parameter which controls how often the noise is caused influenced our results more. Differences were usually up to two positions with three exceptions. Those exceptions represent the ConTest random setting and both versions of our heuristics which in fact do not use the *nFreq* parameter. All three configurations obtain a better ranking when *noiseFreq=50* and non-relativised results are considered. As for relativised results, the ConTest random setting obtained the best overall ranking in both considered evaluation schemes. Our noise injection heuristics used without the newly proposed noise strength computation remained among the best three configurations, still beating the ConTest random configuration in some test results. The combined use of both newly proposed heuristics lost when considering the Avio* coverage, but remained well-ranked when considering the error detection probability. Therefore, we suggest to use the ConTest random setting or our noise injection heuristics without the newly proposed noise strength computation in cases when the amount of noise needs to be very low.

## 5 Conclusions

We have provided a comparison of multiple noise injection heuristics that was missing in the current literature. We have also proposed a new, original noise injection heuristics, winning over the existing ones in some cases. We show that there is no silver bullet among the existing noise injection heuristics although

some of them are on average winning in certain testing scenarios. Based on our experiences, we have given several suggestions on how to test concurrent programs using the noise injection approach. Our future work includes further improvements of our heuristics, a further investigation of the influence of noise on different programs, and an evaluation of some heuristics [4, 6] not yet implemented and tested in our framework. The obtained results are also important for our current work which applies search techniques for automatic identification of a suitable configuration (or configurations) for specific test cases [7, 8].

*Acknowledgement.* This work was supported by the Czech Science Foundation (projects no. 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 project FIT-11-1.

## References

1. Y. Ben-Asher, E. Farchi, and Y. Eytani. Heuristics for Finding Concurrent Bugs. In *Proc. of IPDPS'03*, IEEE CS, 2003.
2. A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*, ACM, 2005.
3. O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41:111–125, 2002.
4. Y. Eytani. Concurrent Java Test Generation as a Search Problem. *ENTCS*, 144, 2006.
5. Y. Eytani and T. Latvala. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. In *Proc. of HVC'06*, LNCS 4383, Springer-Verlag, 2007.
6. E. Farchi, Y. Nir, and S. Ur. Concurrent Bug Patterns and How To Test Them. In *Proc. of IPDPS'03*, IEEE CS, 2003.
7. B. Křena, Z. Letko, and T. Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. To appear in *Proc. of RV'11*, 2011. Available at `http://www.fit.vutbr.cz/~iletko/pub/rv11paper.pdf`.
8. B. Křena, Z. Letko, T. Vojnar, and S. Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*, ACM, 2010.
9. Y. Nir-Buchbinder, E. Farchi, R. Tzoref-Brill, and S. Ur. IBM Contest Documentation, May 2005. `http://www.alphaworks.ibm.com/tech/contest`
10. J. Soriano, M. Jimenez, J. M. Cantera, and J. J. Hierro. Delivering Mobile Enterprise Services on Morfeo's MC Open Source Platform. In *Proc. of MDM'06*, IEEE CS, 2006.
11. S. D. Stoller. Testing Concurrent Java Programs Using Randomized Scheduling. In *Proc. of RV'02*, ENTCS, 70(4), Elsevier, 2002.
12. 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-Verlag, 2011.
13. E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*, ACM, 2009.
14. C.-S. D. Yang, A. L. Souter, and L. L. Pollock. All-DU-Path Coverage for Parallel Programs. In *Proc. of ISSTA'98*, ACM, 1998.