SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2013; **00**:1–38
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Advances in Noise-based Testing of Concurrent Software

J. Fiedor, V. Hrubá, B. Křena, Z. Letko*, S. Ur, and T. Vojnar

FIT, Brno University of Technology, IT4Innovations Centre of Excellence, Czech Republic
Shmuel Ur Innovations Ltd., Israel

SUMMARY

Testing of concurrent software written in programming languages like Java and C/C++ is a highly challenging task due to the many possible interactions among threads. A simple, cheap, and effective approach that addresses this challenge is testing with *noise injection* which influences the scheduling so that different interleavings of concurrent actions are witnessed. In this paper, multiple results achieved recently in the area of noise-injection-based testing by the authors are presented in a unified and extended way. In particular, various *concurrency coverage metrics* are presented first. Then, multiple heuristics for solving the *noise placement problem* (i.e., where and when to generate noise) as well as the *noise seeding problem* (i.e., how to generate the noise) are introduced and experimentally evaluated. In addition, several new heuristics are proposed and included into the evaluation too. Recommendations on how to set up noise-based testing for particular scenarios are then given. Finally, a novel use of the genetic algorithm for finding suitable combinations of the many parameters of tests and noise techniques is presented.

Received . . .

KEY WORDS: testing; dynamic analysis; noise injection; multi-threaded software; coverage metrics; genetic algorithm; meta-heuristic algorithms

1. INTRODUCTION

Popularity of multi-core processors and multiprocessor computers stimulates development of programs with multiple concurrently executing threads of control. Development of such applications in widely used programming languages, like Java, C, and C++, puts much higher demands on programmers who must correctly synchronize actions executed in the different threads communicating via shared memory and/or via message passing. Synchronization errors, such as data races, atomicity violations, deadlocks, or order violations, are relatively easy to cause but very hard to detect by code review or by simple execution of the code during classical testing because they may manifest only under very rare interleavings of actions executed by the different threads. Such interleavings are not very likely to be spot during classical testing, but they can occur in the production where the software is run for a much longer time, on different machines, under different load, and in different environment settings.

This situation in turn stimulates research efforts devoted to all sorts of advanced methods for testing, analysis, and verification of concurrent programs. Formal methods of verification, such

*Correspondence to: Faculty of Information Technology, Brno University of Technology, Božetěchova 1/2, Brno CZ-61266, Czech Republic. email: iletko@fit.vutbr.cz.

Contract/grant sponsor: The work presented here was supported by the Czech Science Foundation (project P103/10/0306), the Czech Ministry of Education (project Kontakt II LH13265), the EU/Czech IT4Innovations Centre of Excellence project CZ.1.05/1.1.00/02.0070 as well as the project CZ.1.07/2.3.00/30.0005, and the internal BUT projects FIT-S-12-1 and FIT-S-14-2486.

as, e.g., model checking [7, 13], may potentially be able to precisely analyze a given program. Unfortunately, these precise approaches do not scale well for complex software systems. The number of thread interleavings to be analyzed in such systems is simply too high to be handled by the precise approaches despite various optimizations that are used in advanced formal verification techniques. Approaches like lightweight static analyses (described below) as well as testing and dynamic analyses (introduced below and further described in the next section) use approximations of the analyzed programs to cope with the complexity of the systems, which can pay off in the number of detected errors despite such approaches can both miss errors as well as produce false alarms [5].

Lightweight static analyses, such as [25], usually focus on searching for purely syntactic error patterns (possibly slightly refined, e.g., by using some information on the behavior of the verified programs pre-computed by suitable dataflow or type analyses). Such analyses scale well to even large code bases and may provide valuable information to the developer [29], but they often cannot discover concurrency-related errors because they do not model threads and their interactions [25]. Of course, there also exist static analyses which do consider concurrent threads, such as, e.g., [6, 44]. These analyses are able to detect concurrency-related errors, but they often produce many false alarms due to the abstractions they work with.

Testing [14, 24, 43, 64] and dynamic analyses [10, 16] rely on (possibly repeated) execution of the given program and evaluation of the witnessed runs. They can precisely analyze all aspects of concurrent behavior, but they only consider the witnessed execution paths and thread interactions (or their extrapolation in the case of dynamic analyses). To increase the number of tested thread interactions, one can use either the deterministic testing approach or noise injection.

Deterministic testing [24, 43, 64] can be viewed as execution-based model checking bounded in various ways (e.g., in the number of context switches), attempting to systematically test as many thread interleaving scenarios as possible. A lightweight alternative to deterministic testing is *noise injection* [14]. This approach is based on injecting—either randomly or based on some heuristics—some noise into the test execution. The noise causes delays in the execution of selected threads, giving other threads an opportunity to make progress and possibly reveal so far untested scheduling scenarios. Although the noise injection approach cannot prove correctness of a program even under some bounds on its behavior, it was demonstrated [14, 36, 57] that the technique can rapidly increase the probability of spotting concurrency errors.

In testing, a crucial role is played by the *coverage metrics* [11]. A coverage metric defines a *coverage domain* which is a set of *coverage tasks* representing different phenomena (e.g., reachability of a certain line) whose occurrence in the behavior of a tested program is considered to be of interest. One can then measure how many of the phenomena corresponding to the coverage tasks have been seen in the witnessed behaviors of the tested program. Such a measurement can be used to asses how well the program has been tested.

Classic coverage metrics (such as code coverage) allow one to relatively easily measure the obtained coverage and to quite precisely estimate the size of the coverage domain statically (up to issues such as unreachable code). However, such metrics do not reflect interactions among threads and are therefore insufficient for testing concurrent programs. Measuring coverage of all thread interleavings, on the other hand, is impractical because one would have to remember and compare various executions from the point of view of all involved context switches, and for an unbounded number of threads, the coverage domain could be unbounded too (for a bounded number of threads, it would be bounded, but huge and difficult to estimate with satisfactory precision). A good concurrency-related coverage metric should hence represent a trade-off between these two extremes.

In this paper, a unified overview of multiple results from the area of noise-injection-based testing that were published by the authors in several past years [20, 21, 26, 32, 33, 34, 36] is provided. In particular, multiple novel coverage metrics (first proposed by Křena et al. [32]), which measure how well the behavior of tested programs has been examined from the point of view of possible occurrence of certain synchronization-related errors, are presented first. These metrics can be used to control saturation-based testing, to compare effectiveness of various testing approaches, or to tune parameters of metaheuristic algorithms applied in testing [26]. Next, various

DOI: 10.1002/stvr

noise injection heuristics and their influence on error detection and ability to increase concurrency-related coverage [20, 34, 36] (absolutely and relatively, i.e., taking into account the longer test execution time caused by noise injection) are studied. The obtained experiences with noise injection on different levels (namely, Java bytecode and C/C++ binaries) are summarized into suggestions that can make further applications of noise-based testing easier. Finally, an application of a metaheuristic algorithm (namely, the genetic algorithm) for an automatic selection of suitable combinations of values of the many parameters of tests and noise injection techniques [26, 33] is presented. The recent works of the authors unified in the paper are accompanied by a description of related works, and the paper can thus serve as a survey of the field of noise-based testing of concurrent software too.

Moreover, in this paper, two new heuristics for noise injection are presented—in particular, a new noise placement heuristic based on access patterns of shared variables and a new noise seeding heuristic which blocks all threads but one. Both of these heuristics target common atomicity violation scenarios, and the newly proposed noise seeding heuristic might also help in order violation scenarios. The newly proposed heuristics are compared with a selection of already existing heuristics which provided promising results in the previous experimental comparisons [20, 36]. The presented set of 8 Java benchmark programs and 4 C benchmark programs makes the comparison the so-far largest comparison of noise-injection-based testing techniques. The comparison shows that the different heuristics can indeed significantly improve the efficiency of testing. However, they also show that there is no single best noise injection technique among the many noise injection heuristics, requiring a careful selection of the noise injection technique to be used in a given scenario and/or a random mix of the heuristics to be applied (possibly aided by metaheuristics as also discussed below).

Plan of the paper. The rest of the paper is organized as follows. In the next section, the state of the art of dynamic analysis and testing of concurrent programs is presented. Concurrency coverage metrics are discussed in Section 3. Section 4 contains an overview of noise placement and noise seeding heuristics, a proposal of the new heuristics, a summary of previously published comparisons, and an evaluation of the newly proposed heuristics. Various technical aspects of noise injection on binary and byte-code levels are also briefly discussed. Finally, a few suggestions for noise-based testing are provided. Section 5 introduces the test and noise configuration problem and shows how the genetic algorithm can be used to solve this problem. Finally, Section 6 concludes the article and several possible future directions in the area of noise-based testing are mentioned.

2. STATE OF THE ART

In this section, a broader overview of the existing techniques for testing of concurrent programs is presented. First, light-weight methods of stress testing and noise injection are discussed followed by methods based on a deterministic scheduler which fully controls the interleaving of actions executed in different threads. Finally, dynamic analysis techniques are introduced and briefly discussed.

Many discussions on various forums suggest to use *stress testing* for discovering concurrency-related errors by simply executing a large number of threads competing for shared resources. This approach increases the possibility of spotting concurrency errors a little, and it can help to reveal some concurrency errors—usually those which manifest quite often. This might make developers to get a false conviction that the program is tested enough [47].

Noise injection inserts delays into the execution of selected threads with the aim of possibly causing new (legal) interleavings, which have so far not been witnessed and tested, to appear. This approach allows one to test more interleavings of synchronization-sensitive actions in shorter time because the system is not that much overloaded by other actions. Noise injection is also able to test legal interleavings of actions which are far away from each other in terms of execution time and in terms of the number of concurrency-relevant events [14] between those actions) during average executions provided that strong enough noise is injected into some of the threads. In

Prepared using stvrauth.cls
DOI: 10.1002/stvr

a sense, the approach is similar to running the program inside a model checker such as JPF [59] with a random exploration algorithm enabled. However, model checkers such as JPF are often limited in the programming constructs they natively support. Moreover, making purely random scheduling decisions may be less efficient than using some of the noise heuristics which influence the scheduling at some carefully selected places important from the point of view of synchronization only. The approach of noise injection is mature enough to be used for testing of real-life software, and it is supported by industrial-strength tools, such as IBM Java Concurrency Testing Tool (ConTest) [14] or the Microsoft Driver Verifier where the technique is called delay fuzzing [1]. Within IBM, ConTest allowed many bugs to be discovered, and as far as we can say, it is still in industrial use.

Deterministic testing, c.f., e.g., [24, 43, 62, 64] has become quite popular recently. The technique uses a deterministic control over the scheduling of threads. A deterministic scheduler is sometimes implemented using intense noise injection keeping all threads blocked except the one chosen for making a progress. Often, other threads which do not execute synchronization-relevant instructions or which do not access shared memory are also allowed to make progress concurrently.

The deterministic testing approach can be seen as execution-based model checking which systematically tests as many thread interleaving scenarios as possible. Before execution of each instruction which is considered as relevant from the point of view of detecting concurrency-related errors, the technique computes all possible scheduler decisions. The concrete set of instructions considered as concurrency-relevant depends on the particular implementation of the technique (often, shared memory accesses and synchronization relevant instructions are considered as concurrency relevant). Each such decision point is considered a state in the state space of the system under test, and each possible decision is considered an enabled transition at that state. The decisions that are explored from each state are recorded in the form of a partially ordered happens-before graph [43], totally ordered list of synchronization events [62], or simply in the form of a set of explored decisions [24, 64]. During the next execution of the program, the recorded scheduling decisions can be enforced again when doing a replay or changed when testing with the aim of enforcing a new interleaving scenario.

As the number of possible scheduling decisions is high for complex programs, several optimizations and heuristics reducing the number of decisions to explore have been proposed. The *locality hypothesis* [43] says that most concurrency-related errors can be exposed using a small number of preemptions. This hypothesis is exploited in the CHESS tool [43] which limits the number of context switches taking place in the execution (iteratively increasing the bound on the allowed number of context switches). Moreover, the tool also utilizes a partial-order reduction algorithm blocking exploration of states equal to the already explored states (based on an equivalence defined on happens-before graphs). The Maple tool [64] limits the number of context switches to two and additionally gets use of the *value-independence hypothesis* which states that exposing a concurrency error does not depend on data values. Moreover, the Maple tool does not consider interleavings where two related actions executed in different threads are too far away from each other. The distance of such actions is computed by counting actions in one of the threads, and the threshold is referred to as a *vulnerability window* [64].

However, despite a great impact of the above mentioned reductions, the number of thread interleavings to be explored remains big for real-life programs and therefore the approach provides great benefit mainly in the area of unit testing [24, 64, 43]. The deterministic testing approach is not as expensive as full model checking, but it is still quite costly because one needs to track which scheduling scenarios of possibly very long runs have been witnessed and systematically force new ones. The approach makes it easy to replay an execution where an error was detected, but it has problems with handling various external sources of non-determinism (e.g., input events).

Deterministic testing offers several important benefits over noise injection. Its full control over the scheduler allows deterministic testing to precisely navigate the execution of the program under test, to explore different interleavings in each run, and to also replay interesting runs (if other sources of nondeterminism, such as input values, are handled). It allows the user to get information about what fraction of (discovered) scheduling decisions has already been covered by the testing process.

DOI: 10.1002/stvr

However, the approach does also suffer from various problems. The approach has problems to deal with external sources of non-determinism (user actions in GUI, client requests) as well as with continuously running programs where its ability to reuse already collected information is limited. In all those problematic cases, noise injection can be successfully used. Moreover, the performance degradation introduced by noise injection is significantly lower.

Another way to improve traditional concurrency testing is to use *dynamic analysis* which collects various pieces of information along the executed path and tries to extrapolate the witnessed behavior in order to find errors which are in the program but did not necessarily occur during the execution. Many problem-specific dynamic analyses have been proposed for detecting special classes of errors, such as data races [16, 48, 49], atomicity violations [38], or deadlocks [10, 28]. These techniques may find more bugs in fewer executions than classical testing. Some of the techniques, e.g., [16], are even sound (i.e., do not miss an error) and precise (i.e., do not suffer from false alarms) with respect to the observed execution path. However, most of the approaches are unsound and typically produce many false alarms.

Efficiency of dynamic analysis can be increased when a different execution path is analyzed during each execution of the test. A combination of noise injection or deterministic testing and dynamic analysis can thus lead to a synergy effect. However, monitoring of the program behavior by a dynamic analysis algorithm typically introduces further synchronization among threads and represents a form of noise affecting thread scheduling, which may be important to take into account when applying regular noise injection heuristics.

Finally, there are tools and techniques that combine various approaches to test multi-threaded programs. For instance, multiple techniques get use of information obtained by static and/or dynamic analysis in navigating deterministic testing tools. An example of such a technique is the recently published *active testing* approach, targeting certain types of errors, such as data races [50], atomicity violations [46], and deadlocks [28]. The technique uses results of approximate static and/or dynamic analyses to hint deterministic testing where a potential error can be found. The technique works in two stages. During the first *prediction phase*, a static and/or dynamic analysis is performed and warnings about specific concurrency errors are collected. In the second *validation phase*, the test is repeatedly executed with a deterministic scheduler. The scheduler behaves as a random scheduler until some thread reaches an action discovered during the prediction phase. If such an action is spotted, all threads that are about to execute this action are stopped. Whenever more threads are stopped, the scheduler enforces all possible interleavings. A similar approach is described in the paper [19] which combines dynamic analysis and bounded model checking. In particular, this approach uses dynamic analysis to detect possible defects in a program and to partially record a behavior witnessing such a defect. An attempt to reconstruct the partially recorded behavior in a model checker is then done using its ability of state space exploration to navigate through the recorded points. Subsequently, bounded model checking in the neighborhood of the behavior can be used to check whether there is really an error in the system or not.

3. CONCURRENCY COVERAGE METRICS AND SATURATION-BASED TESTING

Coverage metrics play a crucial role in testing as they allow one to estimate how well a program has been tested. Based on this information, one can then decide whether to stop the testing process (if the program has been tested enough) or to add new test cases in hope to examine so far uncovered behavior of the program. In sequential programs, repeating the same test case many times on the same version makes no sense as if the test did not find a bug in the first execution, running it again will not help. Of course, automated tests can be used every few days or weeks on different versions of the sequential software which is known as regression testing. In concurrent programs, however, the same tests may be executed multiple times because concurrency errors usually manifest with low probability.

Coverage metrics successfully used for testing of sequential programs (e.g., statement coverage or condition coverage) are not sufficient for testing of concurrent programs as they do not reflect concurrent behavior. When proposing a concurrency coverage metric, one needs to capture

Prepared using stvrauth.cls
DOI: 10.1002/stvr

significant concurrency aspects of the computation by coverage tasks in a way that growing coverage will be related with the potential to reveal concurrency errors. At the same time, it is desirable to neglect worthless facts introduced by a usually huge number of possible interleavings of threads in order to keep the number of coverage tasks reasonable.

3.1. General-Purpose Concurrency Coverage Metrics

In order to measure concurrency-related aspects of software execution, several approaches have extended sequential coverage metrics by capturing interleavings of threads and/or synchronization events [42, 56, 63, 11, 37, 57]. Below, we present three of these metrics, namely, those which were used for an experimental comparison with our new class of concurrency metrics inspired by dynamic detectors of concurrency errors, which have been proposed by Křena et al. [32] and which are discussed in detail in Section 3.2. Compared with the previously proposed metrics, the new metrics are more specialized in tracking behavior considered as important for finding specific classes of synchronization errors by various dynamic analyzers.

Coverage based on concurrently executing instructions (ConcurPairs). The coverage of concurrent pairs of events [11] is a metric in which each coverage task is composed of a pair of program locations that are assumed to be encountered consecutively in a run and a third item that is $true$ or $false$. It is $false$ iff the two locations are visited by the same thread and $true$ otherwise—that is, $true$ means that there occurred a context switch between the two program locations. This metric provides statement coverage information (using the $false$ flag) and interleaving information (using the $true$ flag) at once. A task of this metric is denoted as a tuple $(pl_1, pl_2, switch)$ where pl_1, pl_2 represent consecutive program locations (only concurrency primitives and variable accesses are monitored), and $switch \in \{true, false\}$ indicates whether the context switch occurs in between of them.

Definition-use coverage (DUPairs). This coverage metric is based on the *all-du-path* coverage metric for parallel programs [63]. The metric considers coverage tasks in the form of triples (var, n_u^i, n_v^j) where n_u^i is the u^{th} node in thread T_i where the value of program variable var is defined while it is referenced in v^{th} node in thread T_j. A path in a Parallel Program Flow Graph (PPFG) covers such coverage task if the value of variable var is first defined by thread T_i and then the same value is used in T_j. This can be only guaranteed if a synchronization among threads T_i and T_j taking place between the variable definition and its use. The original approach considers quite simple model of parallel computation, for instance, it supports *post* and *wait* system of synchronization and *pthread_create* operation for creating new threads only, just the master thread is allowed to create worker threads, and the number of created threads in a program need to be determined statically. Under these limitation, it is possible to number the particular threads. When dealing with today real-life applications, one cannot apply such restrictions. The original coverage metric was therefore slightly modified [30]. The modified metric is referenced to as DUPairs* below. The coverage tasks of this metric has the form of tuples $(var, pl_1, pl_2, t_1, t_2)$ meaning that value of variable var is defined at program location pl_1 in thread t_1 and then used at program location pl_2 in thread t_2. Instead of precise numbering of individual threads the metric uses an abstract thread identification which is explained later in this section.

Synchronization coverage (Sync). The synchronization coverage [57] focuses on the use of synchronization primitives and does not directly consider thread interleavings. Coverage tasks of the metric are defined based on various distinctive situations that can occur when using each specific type of synchronization primitives. For instance, in the case of a synchronized block (defined using the Java keyword `synchronized`), the obtained tasks are: *synchronization visited*, *synchronization blocking*, and *synchronization blocked*. The synchronization visited task is basically just a sequential control flow coverage task. The other two are reported when there is an actual contention between synchronized blocks—when a thread t_1 reaches a synchronized block A and

DOI: 10.1002/stvr

stops because another thread t_2 is inside a block B synchronized on the same lock. In this case, A is reported as blocked, and B as blocking (both, in addition, as visited). Tasks of this metric are denoted in this article as tuples of the form $(pl, mode)$ where pl represents the program location of a synchronization primitive, and $mode$ represents an element from the set of the distinctive situations relevant for the given type of synchronization.

3.2. Coverage Metrics Inspired by Concurrency Error Detectors

We are now going to discuss a class of metrics [32] that are more specialized than the metrics above in that they concentrate on concurrency-related aspects of program behavior tracked by various dynamic concurrency error detection techniques, such as Eraser [49], GoldiLocks [16], AVIO [38], or GoodLock [10]. The motivation for this approach comes from two observations: (1) These detection techniques focus on those events occurring in runs of the analyzed programs that appear relevant for detection of various concurrency-related errors. (2) 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 metric to an increasing likelihood of spotting an error.

A crucial step in the creation of a new coverage metric 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 metric. There is a trade-off between precision of the metric and the associated computational complexity. One extreme is to build a coverage metric directly on warnings about concurrency errors issued by the detection algorithm. In this case, the detection algorithm needs to be implemented entirely. Another extreme is to build a coverage metric counting just the events tracked by the detection algorithm and entirely avoid implementation of the algorithm.

Precision of the constructed metrics can further be suitably adjusted by combining their coverage tasks with some *abstract identification of the threads* involved in generating the phenomena reflected in the concerned tasks (extended versions of the metrics are denoted by * in this article). The identification should, of course, not be based on the unique thread identifiers, but it should preserve information on their type, the history of their creation, etc. A similar identification can then also be used whenever the coverage tasks contain some dynamically instantiated objects (e.g., locks). One possible concrete way how the needed identifiers may be obtained is discussed in Section 3.3.

In the text below, the *Java memory model* [40] and the following notation is assumed. V is a set of identifiers of instances of non-volatile variables that may be used in the tested program at hand, O is a set of identifiers of instances of volatile variables used in the program, L is a set of identifiers of locks used in the program, T is a set of identifiers of all threads that may be created by the program, and P is a set of all program locations in the program (i.e., unique identifiers of instructions present in the code or byte-code).

Coverage metrics based on Eraser. The coverage metrics Eraser and Eraser* are based on the Eraser algorithm [49]. For each thread, the algorithm computes a set of locks currently held by the thread, and for each variable access, the algorithm uses these sets to derive the set of locks that were held by each thread that had so far accessed the variable. These locksets are maintained according to a *state* assigned to each variable which represents how the variable has been operated so far (e.g., exclusively within one thread, shared among threads, for reading only, etc.). Basic coverage tasks have the form of a tuple $(pl, var, state, lockset)$ where $pl \in P$ identifies the program location of an instruction accessing a shared variable $var \in V$, $state \in \{virgin, exclusive, exclusive', shared, modified, race\}$ indicates the state in which the Eraser's finite control automaton is when the given location is reached (the extended version of Eraser using the $exclusive'$ state [60] is considered), and $lockset \subseteq L$ denotes a set of locks currently guarding the variable var. Eraser* extends the basic Eraser metric by identification of a thread $t \in T$ performing the access operation. Extended coverage tasks thus have the form of $(pl, var, state, lockset, t)$. Accessing a variable var at a certain program location pl is a code

DOI: 10.1002/stvr

coverage task which is here enriched by the information whether the variable has been already initialized (indicated by $virgin$ or $exclusive$ state). Other possible values of the state cannot be reached in single threaded applications.

Coverage metrics based on GoldiLocks. GoldiLocks [16] is an advanced lockset-based algorithm which combines the use of locksets with computing the happens-before relation that says which events are *guaranteed* to happen before other events. In GoldiLocks, locksets are allowed to contain not only locks (L) but also volatile variables (O) and threads (T). If a thread t appears in the lockset of a variable when the variable is accessed, it means that t is properly synchronized for using the given variable because all other accesses that might cause a data race are guaranteed to happen before the current access. The algorithm uses a limited number of elements placed in the lockset to represent an important part of the synchronization history preceding an access to a shared variable. The basic GoldiLocks algorithm is still relatively expensive but can be optimized by the *short circuit checks* (SC) which are three cheap checks that are sufficient for deciding race freedom between the two last accesses to a variable. The original algorithm is then used only when SC cannot prove race freedom. The basic GoldiLock metric is based on coverage tasks having the form of tuples $(pl, var, goldiLockSet)$ where $pl \in P$ gives the location of an instruction accessing a variable $var \in V$ and $goldiLockSet \subseteq O \cup L \cup T$ represents the lockset computed by GoldiLocks. The tuple can be extended by a thread $t \in T$ which accesses the variable var getting GoldiLock* coverage tasks of the form $(pl, var, goldiLockSet, t)$. Program location pl at which the variable var has been accessed represents a code coverage task. For single threaded applications, one of the short circuit checks discovers that data race cannot occur and the information about execution history captured in $goldiLockSet$ can thus only distinguish the first access to the variable from the others.

Coverage metrics based on Avio. The Avio algorithm [38] detects atomicity violation over one variable and does not require any additional information from the user about instructions that should be executed atomically. The algorithm considers any two consecutive accesses a_1 and a_2 from one thread to a shared variable var to form an atomic block B. Serializability is then defined based on an analysis of what can happen when B is interleaved with some read or write access a_3 from another thread to the variable var. Out of the eight total cases arising in this way, four (namely, r/w/r, w/w/r, w/r/w, r/w/w) are considered to lead to an unserializable execution. Tracking of all accesses that occur concurrently to a block B can be very expensive. Therefore, a criterion to consider only the last interleaving access to the concerned variable from a different thread is defined. The basic Avio metric uses coverage tasks in the form of tuples (pl_1, pl_2, pl_3, var) where the considered atomic block B spans between program locations $pl_1 \in P$ and $pl_2 \in P$ where the variable $var \in V$ is accessed by a thread $t_1 \in T$ while it interferes with the access from a different thread $t_2 \in T, t_2 \neq t_1$ at program location $pl_3 \in P$. The extended metric Avio* incorporates into coverage tasks also information about the threads from which the accesses have been made resulting in tuples of the form $(pl_1, pl_2, pl_3, var, t_1, t_2)$. Single threaded programs cannot generate any such coverage task because basic as well as extended version of Avio-based coverage metric requires the variable var to be accessed by two distinct threads.

Coverage metrics based on GoodLock. GoodLock is a popular deadlock detection algorithm that has several implementations—the metric presented here builds on the implementation published by Bensalem and Havelund [10]. The algorithm builds the *guarded lock graph* which is a labeled oriented graph where nodes represent locks, and edges represent nested locking within which a thread that already has some lock asks for another one. Labels over edges provide additional information about the thread that creates the edge. The algorithm searches for cycles in the graph wrt. the edge labels in order to detect deadlocks. The metrics focus on occurrence of nested locking that is considered interesting by GoodLock. Collection of the locksets of the threads which the original algorithm uses as one element of the edge label is omitted because this information is used in the algorithm to suppress certain false alarms only. The GoodLock metric is therefore based

DOI: 10.1002/stvr

on coverage tasks in the form of tuples (pl_1, pl_2, l_1, l_2) meaning that some thread $t \in T$ has first obtained the lock $l_1 \in L$ at the location $pl_1 \in P$ and later requested the lock $l_2 \in L$ at the location $pl_2 \in P$. The extended metric GoodLock* incorporates also identification of the thread t forming the tuple $(pl_1, pl_2, l_1, l_2, t)$. Locks are usually used for synchronization of accesses to shared resources among several threads, however, also a single threaded application can request for locks and thus generate GoodLock-based coverage tasks.

Coverage metrics based on happens-before pairs. These coverage metrics are motivated by observations obtained from the GoldiLocks algorithm and the vector-clock algorithms [48], both of them depend on computation of the happens-before relation. In order to get rid of the possibly huge number of coverage tasks produced by the vector-clock algorithms and trying to decrease the computational complexity needed when the full GoldiLocks algorithm is used, the metrics focus on pieces of information the algorithms use for creating their representations of the analyzed program behaviors. All of these algorithms rely on synchronization events observed along the execution path. Inspired by this, the metrics capture successful synchronization events based on locks, volatile variables, wait-notify operations, and thread start and join operations used in Java. A basic coverage task is defined as a tuple $(pl_1, pl_2, syncObj)$ where $pl_1 \in P$ is a program location in a thread $t_1 \in T$ that was synchronized with the location $pl_2 \in P$ of the thread $t_2 \in T, t_2 \neq t_1$ using the synchronization object $syncObj$. The extended metric HBPair* incorporates identification of the synchronized threads forming the task as a tuple $(pl_1, pl_2, syncObj, t_1, t_2)$. In the same way as for the Avio-based metrics, no single threaded application can generate any HBPair or HBPair* coverage task because it captures a synchronization between two distinct threads only.

3.3. Abstract Object and Thread Identification

Some coverage metrics described in the previous paragraphs are based on tasks that include identification of threads, instances of variables, and locks. The Java virtual machine (JVM) generates identifiers of objects and threads dynamically. Such identifiers are, however, not suitable for the metrics because (1) in long runs, too many of them may be generated, and (2) matching of semantically equivalent tasks generated in different runs is necessary (may be not precise much, but at least with reasonable precision). The identifiers generated by JVM for the same threads (from the semantical point of view) in different runs will quite likely be different.

Previous works (such as [51]) used Java types to identify threads. Here, type-based identification of elements is considered as too rough. The goal is to create identifiers which distinguish behavior of objects and threads within the program more accurately, but still keep a reasonable level of abstraction so the set of such abstract identifiers remains of a moderate size.

The abstract *object identification* that are considered is based on the observation that, usually, objects created in the same place in the program are used in a similar way. For instance, there are usually many instances of the class `String` in an average Java program, but all strings that are created within invocations of the same method will probably be manipulated similarly. Therefore, an object identifier is defined as a tuple $(type, loc)$ where $type$ refers to the type of the object, and loc refers to the top of the stack (excluding calls to constructors) when the object is created. The record at top of the stack contains a method, source file, and line of code.

The abstract *thread identification* is based on an observation that the type and place of creation are not sufficient to build a thread identifier. Several threads created at the same program location (e.g., in a loop) can subsequently process different data and therefore behave differently. More information concerning the thread execution trace is needed in order to better capture the behavior of threads. Therefore, identifiers in the form of tuples $(type, hash)$ are used. The $type$ denotes the type of the object implementing the thread, and $hash$ contains a hash value computed over a sequence of n first method identifiers that the thread executed after its creation (if the thread terminates sooner, then all methods it executed are taken into account). The value of n influences precision of the abstraction. Of course, when a pool of threads (a set of threads started once and used for several tasks) is used, the computation of the hash value must be restarted immediately after picking a thread up from the pool.

DOI: 10.1002/stvr

3.4. Saturation-Based Testing

As already mentioned above, different executions of a test case usually follow several thread schedules and therefore it make sense to repeat the same test case many times. Unfortunately, it is often very hard or even impossible to precisely enumerate all the coverage tasks imposed by a concurrency metric. Many of the potential concurrency coverage tasks may be infeasible and, thus, reaching a reasonably high coverage during testing may not be possible. This problem can be addressed by *saturation-based testing* [51] which monitors the growth of the coverage and when the coverage stops growing for some time, the testing can be stopped. Further, in *search-based testing* [41], a fitness function driving an optimization algorithm used to control the testing process can be based on the values of a coverage metric.

For metrics used in saturation-based or search-based testing, one can identify several specific properties that they should exhibit. First, within the testing process, the obtained coverage should as often as possible grow for a while and then stabilize. Hence, it should not immediately jump to some value and stabilize on it. On the other hand, it should not take too much time for the coverage to stabilize. Also, to enable a reliable detection of stabilization, the coverage should grow as smoothly as possible, i.e., without growing through a series of distinctive shoulders. Next, in the case of examining an erroneous program by the test that can reveal the bug, the stabilization should ideally not happen before the error is really detected. The evaluation of suitability of existing as well as newly proposed concurrency metrics with respect to these properties [32] is summarized below. In the following, brief description of the experiments highlighting main outcomes is presented.

3.4.1. Experimental Setup For experimental evaluation of concurrency metrics, the platform called SearchBestie [33] was used to set up and execute tests with the IBM ConTest. The ConTest tool provides a facility for bytecode instrumentation and a listeners infrastructure allowing one to create *plug-ins* [30] for collecting various pieces of information about the multi-threaded Java programs being executed as well as to easily implement various algorithms for dynamic analyses. The tool is itself able to collect structural coverage metrics (basic blocks, methods) and some concurrency-related metrics (ConcurPairs, Sync) too. ConTest further provides a noise injection facility which injects the noise into the execution of a tested application. Our experiments have been done on five concurrent programs described below.

The *Dining philosophers* test case is an implementation of the well-known synchronization problem of dining philosophers. The program generates a set of 6 philosophers (each represented by a thread) and the same number of shared objects representing forks. A deadlock can occur when executing the test case.

The *Airlines* test case is a simple artificial program consisting of 8 classes and simulating an air ticket reservation system. It generates a database of air tickets and then allows 2 dealers (each represented by a separate thread) to sell tickets to 4 sets of 10 customers (each set is represented by a separate thread). Finally, a check whether the number of customers with tickets is equal to the number of sold tickets is done. The program contains a high-level atomicity violation whose occurrence makes the final check fail.

Crawler is a skeleton of 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 is established, a worker thread serves it. A bug present in the program can cause a deadlock when the crawler shuts down, however, it is seen very rarely (6 times per 10 000 runs). The Crawler test case consists of 19 classes.

Another real-life application among our case studies is an early development version of an open-source *FtpServer* produced by Apache. The version of the server used here contains 120 classes. The server creates a new worker thread for each new incoming connection to serve it. The code contains several data races that can cause exceptions during the shutdown process when there is still an active connection. The probability of spotting an error when noise injection is enabled is quite high in this example because there are multiple places in the test where an exception can be thrown.

Our biggest case study with 1399 classes is *TIDOrbJ*—a CORBA-compliant ORB (Object Request Broker) product that is a part of the MORFEO Community Middleware Platform [52]. The test used checks how the infrastructure handles multiple concurrent simple requests. The

DOI: 10.1002/stvr

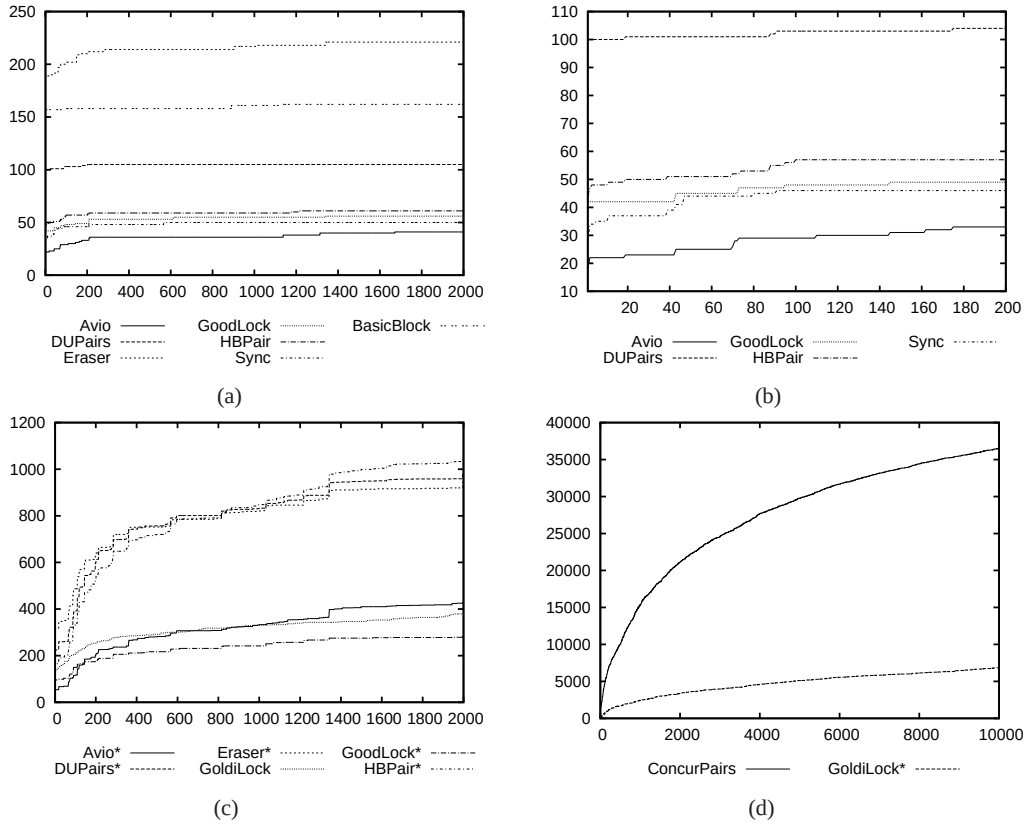

Figure 1. Cumulative values of coverage metrics on the Crawler test case (the horizontal axis gives the number of executions, the vertical axis gives the cumulative number of covered tasks)

particular test of TIDOrbJ we consider starts an instrumented server and then 10 clients, each sending 5 requests to the server. There was originally no error in this test, however, a high-level atomicity violation that leads to a null pointer exception has been injected there by commenting one `synchronized` statement in the code.

10,000 executions of the small programs and 4,000 executions of TIDOrbJ were performed. In order to see as many different legal interleaving scenarios as possible, the ConTest tool was set to randomly inject noise into the executions. ConTest plug-ins to collect coverage information were implemented and SearchBestie was set up to detect occurrences of errors.

3.4.2. Results of Experiments The results of the experiments [32] are shown in Figure 1. All four sub-figures show the cumulative number of coverage tasks of the metrics covered during one randomly chosen series of the Crawler test case executions. The metrics marked with an asterisk are extended by the abstract thread identification (with variable n set to 20).

Figure 1(a) shows the behavior of the metrics that do not capture the concurrent behavior accurately enough. One coverage metric for non-concurrent code measuring the number of *basic blocks* covered during tests is added to demonstrate the difference between classical and concurrency-related coverage metrics. The coverage obtained under the metric based on basic blocks is nearly constant all the time because the same code was executed with the same inputs. For the rest of the metrics shown in Figure 1(a), the cumulative number of tasks covered during test executions increases only within approximately the 200 first executions (zoomed in Figure 1(b)), and then a saturation is reached. The depicted curves demonstrate one further disadvantage of the concerned metrics—a presence of distinctive shoulders. A repeated execution of the test case does examine different concurrent behaviors (which is indicated by the later discussed metrics) but the metrics concerned in the figure are not able to distinguish differences in these behaviors, and therefore

Prepared using stvrauth.cls
DOI: 10.1002/stvr

clear shoulders (i.e., sequences of constant values) can be seen in the curves. The presence of such shoulders makes automatic saturation detection harder.

Figure 1(c) demonstrates a positive effect of considering an extended context of the tracked events as mentioned in Section 3.3. The metrics concerned in this sub-figure (i.e., Avio*, Eraser*, DUPairs*, HBPair*, GoodLock*, and GoldiLock) are able to distinguish differences in the behavior of the executed tests more accurately, leading to shorter shoulders, bigger differences in the cumulated values, and a later occurrence of the saturation effect—indicating that the concerned metrics behave in a way much better for saturation-based testing. A very positive behavior has the GoldiLock metric which does not suffer from shoulders while it reaches saturation near the saturation points of the other metrics. Figure 1(d) shows problems of metrics that are too accurate, namely, ConcurPairs and GoldiLock*. These metrics work fine for small test cases but when used on a bigger test case they tend to saturate late and produce enormous numbers of covered tasks.

To summarize the results, the most favorable behavior for saturation-based testing shows the GoldiLock metric. Very good behavior have also Avio*, Eraser*, DUPair*, HBPair*, and GoodLock* metrics. The GoodLock* metric can provide information which cannot be directly inferred from any other examined metric. On the other hand, if nested locking does not occur in the tested program, the GoodLock* metric provides no information. The metrics Avio, GoodLock, HBPair, Eraser, and Sync do not capture the concurrent behavior accurately enough while ConcurPairs and GoldiLock* metrics are too accurate (they work fine for small test cases but when used on a bigger test case they tend to saturate late and produce enormous numbers of covered tasks).

4. NOISE INJECTION TECHNIQUES

As already mentioned in the introduction, the effectiveness of noise-injection-based testing techniques depends on a satisfactory solution to the *noise placement* and *noise seeding* problems. The noise placement problem addresses the question where, i.e., at which program locations, and when, i.e., at which executions of these locations, to cause a noise. The noise seeding problem then determines how to cause the noise, i.e., which type of noise generating mechanism should be used, and how long it should last. The problems are, of course, not independent, and so, a suitable *combination* of noise placement and noise seeding heuristics (and of suitable values of their many parameters) is to be sought in practice.

In this section, an overview and comparison of various existing approaches to solving the noise placement and noise seeding problems is presented, including a few newly proposed heuristics for solving these problems. Moreover, a discussion of differences in applying noise-based testing for C/C++ and Java programs is provided. This discussion provides several hints on how to use noise-based testing as well.

This section is organized as follows. In the first two subsections (Section 4.1 and Section 4.2), existing noise placement and noise seeding heuristics are described. Additionally, several new heuristics are introduced. Then, in Section 4.3, a selection of results of previously published comparisons [34, 36, 20] of the older noise injection heuristics are presented. Based on the results, the most promising heuristics and their parameters are pinpointed. Then, a new comparison of the old and newly proposed noise injection heuristics on a set of C and Java benchmarks of various size is provided. The used set of benchmarks is the so-far biggest set of benchmarks used for evaluating noise-injection-based testing. The mentioned selection of the most promising heuristics that we have been done allowed us to test the selected promising noise configurations much more thoroughly (which would not be possible with all of the possible noise configurations due to the high time requirements of the experiments). The obtained results are discussed separately for C/C++ (Section 4.4) and Java (Section 4.5) because of the different noise injection infrastructures used for testing programs written in these languages—in particular, for C/C++ code, instrumentation on the binary level was used, whereas for Java, instrumentation on the bytecode level was used. An insight into the differences of the considered noise-based testing of C/C++ programs on the binary level and of Java programs on the bytecode level is presented in Section 4.6. Despite the differences,

DOI: 10.1002/stvr

a discussion of commonalities and dissimilarities of the obtained results is provided in Section 4.7. Finally, some hints on how to effectively use noise-based testing are presented in Section 4.8.

4.1. Noise Placement Heuristics

Noise placement heuristics determine where, i.e., at which program locations, and when, i.e., at which executions of these locations, should a noise be injected. In this section, an overview of existing noise placement heuristics is provided first and then several novel heuristics addressing the problem of where to put a noise are introduced.

It is discussed in several papers [15, 18, 54] that putting a noise at every possible program location (*ploc* [15]) is inefficient. This approach significantly increases the incurred overhead, and it does not help much in increasing chances to find bugs since only a few relevant context switches are critical for a concurrency error to manifest. Also, it turns out that putting a noise at a certain program location can help to spot the concurrency error, but it can also mask it completely.

The IBM ConTest tool [14] allows one to inject a noise only before and/or after concurrency-related events (namely, accesses to class member variables, static variables, and arrays stored in the JVM heap, calls of `wait()`, `interrupt()`, `notify()`, `monitorenter`, and `monitorexit` routines). Since the tool has no information which member fields and arrays are really shared (i.e., accessed by multiple threads), all instructions operating with the heap are considered. Moreover, motivated by a coding anti-pattern in which developers use calls of `wait()` instead of proper synchronization, ConTest is able to intercept calls to the `wait()` and `sleep()` routines too.

The *rstest* tool [54] considers as possibly interesting only those locations that appear before concurrency-related events. Moreover, rstest uses a simple escape analysis and a lockset-based algorithm to identify the *unprotected accesses* to shared variables. An unprotected access reads or writes a variable which is visible to multiple threads without holding an appropriate lock. This optimization reduces the number of program locations where the noise can be put but suppresses the 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.

Moreover, the number of accesses to shared memory and calls of synchronization elements is still high in multi-threaded programs. Therefore, several heuristics for determining more concretely where and when to put a noise were proposed [9, 15, 54, 57].

The simplest heuristic is based on a random number generator [15, 54]. This *random* heuristic puts a noise before an executed program location with a given probability, where the probability is the same for all program locations considered. Most other heuristics extend this heuristic in a way that they reduce the number of possible program locations before which the noise might be injected. When considering all possible program locations in a program, this heuristic is called *random-all* below to distinguish it from the other heuristics that can be seen as modifications of the *random* heuristic.

It was shown [9] that restricting the number of program locations only to those accessing shared variables or a specific shared variable when applying the *random-all* heuristic increases the probability of spotting an error. These two modifications of the *random-all* heuristic are denoted here as *sharedVar-all* and *sharedVar-one*, respectively. When the *sharedVar-one* heuristic is used, the shared variable is usually chosen randomly from a list of known shared variables.

Several heuristics based on concurrency coverage models have been published. Coverage-directed generation of interleavings [15] 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 can be in most of the cases too coarse. In [57], a coverage model considers, for each synchronization primitive, various distinctive situations that can occur when the primitive is executed (e.g., in the case of a synchronized block defined using the Java keyword `synchronized`, the tasks are: *synchronization visited*, *synchronization blocking* some other thread, and *synchronization blocked* by some other thread). The approach then injects a noise at corresponding synchronization primitive

DOI: 10.1002/stvr

program locations 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.

A coverage-based noise placement heuristic [36] (referred to as *coverage* in this article) targets both accesses to shared variables as well as the use of synchronization primitives, and so it can be used to discover lock-based deadlocks as well as data-related concurrency errors, such as data races and atomicity violations. The heuristic considers only program locations that appear before concurrency-related events as suitable for noise injection. The technique detects subsequent accesses to shared variables and monitors whether these accesses originated in different threads. Such couples of subsequent accesses are considered as interesting to be influenced by noise. The noise in particular tries to test the opposite orderings of recorded events in each couple. Therefore, a noise is put before the first access recorded in a couple with a hope that the access which was recorded as the subsequent occurs earlier. If both accesses are guarded by the same lock, the described approach would inject a noise into a shared critical section which would not change the ordering of the recorded events. In such a case, the heuristic injects the noise before the appropriate locking operation where the common lock was obtained. Additionally, this heuristic monitors the frequency of a program location execution during a test and puts a noise at the given program location with a probability biased wrt. this frequency—the more often a program location is executed the lower probability is used.

Next, a noise placement heuristic called *read/write* heuristic [20] uses different noise settings for the shared variable read accesses and write accesses. The settings might differ in the frequency which controls how often a noise is generated before a particular class of accesses or in the chosen noise seeding heuristic. The heuristic is motivated by the common data race scenarios where there are two unsynchronized accesses to a shared variable and at least one of these accesses is a write access. So, when a memory access is encountered, the best thing one can do is to search the other threads for the second (conflicting) access. In order to lower the noise injected to the other threads, the fact that one of the accesses causing a data race is typically a write access while the other is a read access is exploited. Based on this observation, a stronger noise before one type of memory accesses and a weaker noise before the other is injected.

A new noise placement heuristic. As the *read/write* noise placement heuristic proved to be very useful when detecting data races, a new noise placement heuristic called a *pattern* heuristic is proposed here. The heuristic injects a noise before accesses to variables which were already accessed before within the same method or function. The motivation here is to create a noise placement heuristic that would help in discovering atomicity violation scenarios. An atomicity violation occurs when two accesses to a shared variable, which should be performed atomically, are interleaved by another access to this variable. The idea is to inject a noise before the second (or any further) access to a shared variable from the same thread within a logical unit (a program method in this case) so other threads have more time to access this variable in between the accesses, causing an atomicity violation. It makes sense to inject such noise even inside a block intended by the programmer to be executed atomically (e.g., a block defined by the `synchronized` keyword in Java) and test whether the synchronization is implemented correctly.

4.2. Noise Seeding Heuristics

Noise seeding heuristics determine how to cause a noise, i.e., which type of noise generating mechanism should be used, and how long should it last, i.e., how strong the noise should be. In this section, an overview of existing noise seeding heuristics is provided and then a new heuristic addressing the problem how to cause a noise is proposed.

As the primary purpose of injecting a noise is to disturb the usual scheduling of threads, the noise generating mechanism should influence the scheduler in some way. There exist several ways how a scheduler decision can be affected in Java [15]. The *priority* heuristic changes priorities of threads which allows chosen threads to make progress more often than threads with the lower priority. The *yield* heuristic injects one or more calls of the `yield()` method which causes a context switch. The *sleep* heuristic injects one call of `sleep()`, and the *wait* heuristic injects a call of `wait()`.

DOI: 10.1002/stvr

The `sleep()` and `wait()` methods take a timeout for which to block a thread as their parameter. In case of the *wait* heuristic the concerned thread must first obtain a special shared monitor, then call `wait()` with a timeout on it, and finally, release the monitor. Using the monitor makes the current thread flush all local data to shared memory and make them visible for other threads. Likewise, the *synchYield* heuristic combines the *yield* heuristic with obtaining a monitor. The *busyWait* heuristic does not obtain a monitor, but instead loops for some time.

The *haltOneThread* heuristic [57] occasionally stops one thread until all remaining threads cannot make any further progress. Finally, the *timeoutTamper* heuristic randomly reduces the time-out used when calling the `sleep()` and `wait()` methods in the tested program. This allows one to test that the delay inserted by these methods is not used instead of proper synchronization.

All the noise seeding heuristics mentioned above are parameterized by the *strength of noise*. In case of the *sleep*, *wait* and *busyWait* heuristics, the strength gives the time to wait or loop. In the case of the *yield* heuristic, the strength says how many times the `yield()` routine should be called. Finally, in the case of the *priority* heuristic, the strength determines how much the thread priority changes.

The *barrier* scheduling heuristic [9] based on semaphores is presented. Each shared variable is assigned a specific semaphore in such a way that a thread is made to wait just before the particular shared variable is accessed. When more than one thread is waiting at the same monitor (and thus for access to the same variable), then the `notifyAll()` method is used to simultaneously advance the waiting threads in hope to spot a data race. To prevent deadlocks, the waiting of threads on the injected semaphores is timed.

All the above works discuss noise seeding heuristics for Java. The first results obtained by implementations of the selected noise seeding heuristics, in particular, the *yield* and *sleep* heuristics, in C [20] have been recently presented as well.

A new noise seeding heuristic. Some kinds of concurrency errors manifest in situations where a thread executes an action earlier that it should, e.g., sends a notification before someone starts waiting, accesses a variable before it is initialized, etc. A new noise seeding heuristic called a *inverseNoise* heuristic proposed here does the opposite of the *haltOneThread* heuristic. That is, it stops all but one thread and allows this one thread to get as far as possible in its execution. This increases chances that the thread will trigger an action which it should perform only after some of the blocked threads do something, e.g., start waiting, initialize a variable, etc. Moreover, the other threads are stopped at the nearest instrumentation point which is suitable for noise injection. Therefore, the current thread has the opportunity to execute instructions which trigger an atomicity violation if some of the blocked threads are blocked within an improperly guarded atomic section.

4.3. Results of Previously Performed Comparisons

In this section, the most important aspects of previously published comparisons of noise injection heuristics [20, 34, 36] are highlighted. These results were used to set up an environment for the comparisons presented in this article. In particular, the results lead to a choice among the many possible configurations of noise placement and noise seeding heuristics—those which provide good results in the comparisons presented in this section.

An extensive and systematic comparison of results of various existing noise placement and noise seeding heuristics including the coverage-based noise placement heuristics and the related noise seeding heuristics introduced above for Java has been published in [34]. The heuristics were compared according to their efficiency to improve detection of concurrency errors, to improve the concurrency-related coverage metrics HBPair* and Avio* in the considered test cases, and to affect the execution time of the considered test cases. The HBPair* and Avio* metrics described in Section 3.2 have been chosen due to their very good ratio of providing satisfactory results from the point of view of suitability for saturation-based or search-based testing and a relatively low overhead of measuring the achieved coverage (and hence their suitability for performing many tests with an acceptable interference with the tested programs). The SearchBestie platform [33] was used to set up and execute the needed tests with IBM ConTest [14]. The heuristics were evaluated on

Prepared using stvrauth.cls
DOI: 10.1002/stvr

a set of 4 test cases (namely, Airlines, Crawler, FTPServer, and TIDOorbJ test cases) which have been already described in Section 3.4.1.

First, there was done a comparison of several noise seeding heuristics denoted as basic below (namely, the *yield*, *synchYield*, *wait*, *busyWait*, and *sleep*) and the IBM ConTest *mixed* noise seeding heuristic which randomly chooses one of the basic noise seeding heuristics at each call of the noise injection routine. Then, the improvement which can be achieved by combining basic noise seeding heuristics with the *haltOneThread* and *timeoutTampering* heuristics was studied. All heuristics were used with the *random-all* noise placement heuristic enabled.

The results indicate that there is no optimal configuration, i.e., for each test case and each testing goal (improvement of coverage, error manifestation, or overhead minimization), one needs to choose different noise seeding heuristic [34]. Moreover, in some cases, the noise injection heuristics improved the obtained results considerably while, in some other cases, the noise seeding configurations used with the *random-all* noise placement heuristic actually provided considerably worse—demonstrating the ability of noise injection techniques to mask concurrency errors [31]. The *timeoutTamper* heuristic provided a considerable improvement for the *crawler* test case. As already said, this test case is a skeleton of an IBM software product. When developers extracted the skeleton, they modeled its environment using timed routines. The *timeoutTamper* heuristic influences these timeouts in a way leading to a significantly better results.

Next, a comparison of different noise placement heuristics has been published by Letko [34] as well. Mainly, the *random-all*, *sharedVar*, and *coverage* heuristics were considered. Additionally, a heuristic which randomly sets up noise settings before each test execution was considered in the comparison too. The noise placement heuristics were again compared according to the ability to detect concurrency errors and to provide a high coverage. Then, a comparison of the heuristics using relative results was provided as well. In this comparison, the total number of covered tasks or detected errors was divided by the execution time (in seconds) the heuristics needed to achieve the results.

Again, none of the heuristics achieved best results in the comparisons for all the considered test cases. Overall good results were obtained by different versions of the *sharedVar* heuristic which focuses noise to shared variables only. There was no winner among the two versions of the heuristic: *sharedVar-all* which targets all accesses to shared variables and *sharedVar-one* which targets accesses to a single randomly chosen shared variable in each test execution. The heuristic using random settings for each test execution achieved on average good results too. This was because accumulated results from multiple runs (namely, 20 and 50 times) were used for the comparison—some of the randomly chosen settings therefore provided very good results regardless of the test case which turned in the overall results. The *coverage* heuristic achieved good results in some cases as well.

Finally, the best relative improvement achieved by noise-based testing in the considered test cases was presented by Letko [34]. Table I shows the results obtained when evaluating the best relative improvement (denoted as *Impr.*) in the experiments for the considered metrics and test cases. The improvement is computed as a relative improvement compared to the configuration without noise injection (note that collection of coverage information and the instrumentation itself already introduce a certain amount of noise). The next three columns (denoted as *nFreq*, *Seeding heur.*, and *Placement heur.*) present the noise frequency, noise seeding heuristic, and noise placement heuristic used. Combinations of the basic noise seeding heuristics with the *timeoutTampering* heuristic (denoted as *tt*) and *haltOneThread* heuristic (denoted as *ht*) were also allowed and evaluated.

The improvement of the error manifestation ratio (denoted as *Error*) in the TIDOorbJ test case is not present because the version of the test case we used contains no error. The ★ symbol in the error manifestation ratio of the Crawler test case means that the improvement cannot be computed because in the experiments, the error does not manifest when the noise was disabled. The best value, which was achieved by the *coverage* heuristic, reached 2 % of error manifestation in this test case (on average 1 error manifestation per 50 executions).

In some cases (e.g., in the Airlines test case), the improvement of the error detection is high, reaching several hundreds percents. The lowest improvement was achieved in the FTPServer test

DOI: 10.1002/stvr

Table I. The best relative improvement achieved by noise heuristics

Test	Metric	Impr.	nFreq.	Seeding heur.	Placement heur.
Airlines	Error	5.93	150	yield + tt	sharedVar-one
	Avio*	1.99	–	–	no noise
	HBPair*	1.90	–	–	no noise
Crawler	Error	★	–	busyWait	coverage
	Avio*	8.20	50	mixed + tt + ht	sharedVar-all
	HBPair*	3.55	200	mixed + tt + ht	sharedVar-all
FTPServer	Error	1.09	50	sleep	sharedVar-one
	Avio*	1.26	50	wait + tt + ht	sharedVar-all
	HBPair*	1.55	150	busyWait + ht	sharedVar-all
TIDOrbJ	Error	–			
	Avio*	1.12	200	busyWait + tt + ht	sharedVar-one
	HBPair*	1.23	200	busyWait + tt + ht	sharedVar-one

Table II. Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise injection configuration					Test case		
ConfID	Placement heur.	Seeding heur.	Freq.	Strength	t05	t06	t07
instrumented, no sleep or yield noise					0.0	1.0	1.6
1	random-all	sleep	500	10	1.2	53.6	69.4
2	random-all	sleep	500	0–10	0.6	31.0	79.0
3	read/write	sleep / sleep / sleep	500	10 / 5 / 20	**43.0**	**92.6**	**96.2**
4	read/write	yield / yield / sleep	500	10 / 10 / 10	**51.0**	**95.0**	**99.6**

case. This is mainly because the error manifestation ratio is quite high even without the noise injection and by the fact that any performance degradation in effect makes the code containing the error execute less often. Overall, the table presents the positive effect of relatively cheap and easy to use noise injection technique in the process of testing concurrent programs. Again, one cannot claim a clear winner among the noise placement and noise seeding heuristics. However, the *sharedVar* noise placement heuristic achieved very good overall results in this evaluation.

Next, a comparison of the *read/write* noise placement heuristic with the *random-all* heuristic on a set of 14 C programs implementing a simple ticket algorithm using the pthreads library is presented [20]. These programs were created by students of an advanced operating systems course and all contain data races. They are referred as test cases `t01` to `t14`. The ANaConDA framework [21] was used to perform the tests. The framework uses the Intel PIN framework [39] for dynamic binary instrumentation to insert the code implementing the noise injection heuristics into a C/C++ program binary. As the framework cannot provide concurrent coverage information yet, an evaluation of the successfully detected data races in each test run was performed. For the detection of data races, a C++ implementation of the AtomRace dynamic detector [35] was used.

Results obtained for some selected noise injection configurations and test cases are shown in Tables II and III. Each configuration is defined by a noise placement and noise seeding heuristics together with the values of frequency and strength used (denoted as *Placement heur.*, *Seeding heur.*, *Freq.*, and *Strength*, respectively). If the *read/write* noise placement heuristic is used, the *Seeding heur.* and *Strength* columns then contain 3 values. These are the values used for the synchronization operations, read accesses and write accesses, respectively. In case of the *Seeding heur.* column, the values represent the noise seeding heuristic used, and in case of the *Strength* column, the value of strength used. If the value of strength is an interval, the particular value was taken randomly from the interval each time the noise was injected.

Prepared using stvrauth.cls
DOI: 10.1002/stvr

Table III. Success ratio of the AtomRace detector for various configurations of the noise injection (the values represent the percentage of runs, out of 500, in which a data race was found)

Noise injection configuration					Test case	
ConfID	Placement heur.	Seeding heur.	Freq.	Strength	t04	t05
instrumented, no sleep or yield noise					1.2	0.0
5	read/write	sleep / sleep / yield	100	10 / 10 / 10	7.4	**62.4**
6	read/write	sleep / yield / sleep	100	10 / 10 / 10	**96.8**	9.6
7	read/write	yield / sleep / yield	100	10 / 10 / 10	6.2	**64.4**
8	read/write	yield / yield / sleep	100	10 / 10 / 10	**94.4**	7.2

The *read/write* noise placement heuristic allows to use different noise seeding heuristics and their parameters for different types of memory accesses. Of course, there are many possibilities how to combine them, so the two most promising combinations were focused. First, the same noise seeding heuristics have been used, but parametrized them with different values of strength, i.e., a bigger strength for one type of memory accesses and a considerably lower for the second one was applied. The goal was to lower the amount of noise injected to the threads that are intended to be search through when detecting data races. As the results in Table II show, such configurations (Configuration no. 3) achieved better results than the configurations using the *random-all* heuristic (Configurations no. 1 and 2).

Configurations which use different noise seeding heuristics for different memory accesses were also used. More precisely, the *sleep* heuristic for one type of memory accesses and the *yield* heuristic for the second one were studied. Their values of strength were left the same. The goal was not only to lower the amount of noise injected to the threads to be searched through, but also to allow the threads to perform as many memory accesses as possible. While the sleep noise is blocking the thread performing the first access, the yield noise is forcing the program to quickly switch threads so the threads will be running more often and hence perform more memory accesses. As the results in Table II show, such configurations (Configuration no. 4) achieved even better results than the ones combining different values of strength (Configuration no. 3).

The tests also proved that it is important to choose the right type of memory accesses before which the stronger noise is injected. When there are only a few unprotected write accesses which might cause a data race, the stronger noise should be put before these accesses. This is because it is far more probable that one will encounter the more common read accesses in the other threads which are being searched than the rare write accesses. If the situation is opposite, the stronger noise should be put before the read accesses. Table III shows the difference in results for two programs which mainly differ in how a data race might manifest. As the `t04` test case contains only a few unprotected write accesses which might cause a data race and many unprotected read accesses, the configurations injecting a stronger noise before the write accesses (Configurations no. 6 and 8) give far superior results than configurations injecting a stronger noise before the read accesses (Configurations no. 5 and 7). In case of the `t05` test case which contains only a few unprotected read accesses and many unprotected write accesses, the results are completely opposite.

4.4. A Comparison of Noise Injection Techniques in C/C++

New experiments that were performed with C programs and noise injection heuristics selected according to the experience from older experiments described in the previous section are presented here (new experiments with Java programs will be described in the following section). First, a description of the testing environment and experiment settings which are used to compare selected noise injection heuristics, including the newly proposed heuristics, is given. Then, the obtained results of these heuristics on four C programs are provided.

4.4.1. Testing Environment The framework ANaConDA [21] already mentioned in Section 4.3 was used to perform the tests. For each execution of a test, the framework collects information about test duration and about the fact whether an error is manifested. In contrast to the previous comparisons

DOI: 10.1002/stvr

where each noise configuration was given an equal number of test executions, in this comparison, each considered configuration of noise heuristics was given 20 minutes of real time to test the program and average results were computed. Therefore, the configurations with higher impact on the performance were provided with lower number of executions of the test. This allows to demonstrate efficiency of the heuristics in practical testing scenarios where the time and other resources for testing are usually limited.

As there are many possible combinations of various noise placement and noise seeding heuristics and as each of these heuristics might be parametrized in many different ways, there exists a large number of configurations that might be used. In order to keep the number of considered configurations on a reasonable level, the focus is devoted to heuristics and their parameters which provided good results in the previous comparisons and also on the new heuristics introduced in the previous sections.

In case of the noise placement heuristics, the following ones are considered: the *random-all* heuristic which is used as a base-line, the *sharedVar-all* and *sharedVar-one* heuristics which provided good results in the evaluation of noise placement heuristics for testing Java programs, the *read/write* heuristic which turned out to be efficient in the previous experiments with noise injection in C/C++, and the newly proposed *pattern* heuristic. All these heuristics decide whether to inject a noise based on the *frequency* parameter which controls how often the noise is injected at the selected place. The frequency parameter was set such that the noise was generated either in 15 % or 30 % of situations. These values were also inspired by the results of the previous comparisons.

As for the noise seeding heuristics, the *sleep*, *yield*, and *busyWait* heuristics were considered because they provided good results in some cases in the previous comparisons. Moreover, the newly proposed *inverseNoise* heuristic was added. The noise seeding heuristics are parametrized by the *strength* parameter. This parameter was set to 2 and 20 milliseconds in the case of *sleep* and *busyWait* heuristics and to 10 and 100 executions of the `yield()` function in the case of the *yield* heuristic. In the case of the *read/write* heuristic, the strength parameter for writes and reads was set in the mutually complementary way. That is, if a higher value for writes (e.g., 20 ms) was used, the lower value for reads (i.e., 2 ms) was applied, and vice versa. As for the newly proposed *inverseNoise*, the parameter was set to 2 and 20 operations executed by the current thread while other threads are blocked. The higher values were chosen based on the results of the previous comparisons where a stronger noise often helped more than a weaker one. The lower values were used primarily because of the *read/write* heuristic, where combining strong and a much weaker noise led to the best results. Also, as the *yield* heuristic disturbs the usual scheduling of threads far less than the other noise seeding heuristics, higher values of strength were used for it. In case of the *read/write* noise placement heuristic, configurations combining the *sleep* and *yield* noise seeding heuristics with fixed values of strength were also used (10 for the *sleep* heuristic and 50 for the *yield* heuristic).

The combinations of heuristics described above give 81 noise configurations (5×2 noise placement heuristics, 4×2 noise seeding heuristics, and 1 configuration without noise—referred to as *nonoise* below). Note that the previous comparisons did not contain the *sharedVar-all*, *sharedVar-one* and the newly proposed *pattern* noise placement heuristics. Also, the *busyWait* and the newly proposed *inverseNoise* noise seeding heuristics were not considered. They are thus examined for C/C++ programs for the first time.

For the experiments, 4 simple C programs (about 200 to 500 lines of code) implementing a simple ticket algorithm using the pthreads library were used. These programs were chosen from a set of programs [20] which were already mentioned in Section 4.3. The chosen programs are referred to as test cases `t01`, `t03`, `t05` and `t06`. The main reason to use only a subset of programs was that some of the newly tested noise placement heuristics need information about the variables which are accessed. In case of C/C++ programs, these information need to be extracted from the debugging information of the program. However, the ANaConDA framework has only a partial support for extracting this kind of information, and for many of these programs the compiler generated debugging information which the framework was not able to process. So in order to test these new heuristics, availability of this information is required. As the framework imposes a huge slowdown on the execution of the tested program, bigger programs were not considered for

DOI: 10.1002/stvr

the tests because one would be able to perform only a few testing runs in the given 20 minute time slot. All the programs were executed on an 4-core Intel Xeon X5355 2.66GHz machine with the Hyper-threading support (up to 8 threads might run simultaneously) and 64GB memory running Linux with the 2.6.32 kernel.

The selected programs contain various kinds of errors that all lead to data races in the end. In `t01`, the data race is on a shared variable holding the number of a ticket allowed to enter a critical section. The variable is updated in a critical section, but then read outside of it. The next program (`t03`) contains a data race on a shared variable used to assign IDs to each of the threads. This variable is updated and read without any synchronization, however, all of these accesses happen when the threads are started one immediately after another, so the data race may only occur during this short time. Program `t05` has a rarely occurring data race on individual items of a shared array where each item may be accessed by the main thread and one of the other threads simultaneously just before the main thread starts to wait for the second thread to end (join). Program `t06` contains a data race on a `timespec` structure, shared among all threads, used to randomly generate the number of milliseconds a thread should sleep before and after entering the monitor.

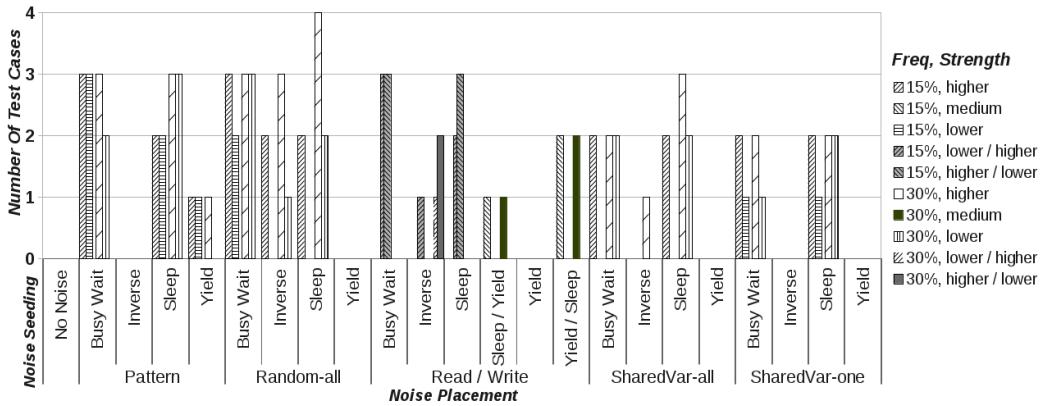

Figure 2. A comparison of configurations across all of the considered C test cases

4.4.2. Experimental Results In this section, a comparison of the efficiency of detecting concurrency-related errors using various noise injection configurations is described first. Then, focus is devoted to the results obtained by the newly proposed heuristics.

Since all of the test cases contain a data race and the consequences of these data races are not always externally visible, a dynamic analysis using the AtomRace dynamic detector [35] was performed in order to find these errors. Like the noise injection, the dynamic analysis requires the program to be instrumented, so it is problematic to compare the results obtained with and without dynamic analysis. However, note that the tests that were originally used to evaluate the considered student projects from which the test cases are derived did not found any errors. The instrumentation of a program usually increases the probability of finding an error, even when no noise is injected, as the execution of the instrumented code itself causes a sort of a very weak noise which might help a little with the error detection. So, even with the *nonoise* configuration, it was possible to detect some errors in the 20 minute time slot in most of the test cases (namely, `t01`, `t03`, and `t06`).

To compare the efficiency of each configuration, their general success across all of the test cases executed was measured. The results are summarized in Figure 2. The x-axis shows the noise configurations grouped by the noise placement and noise seeding heuristics with the values of noise frequency and strength represented by the different hatch of the bars. The y-axis then shows the number of test cases (out of 4) for which the respective configuration was among the best 30 % of the configurations (i.e., among the best 24 configurations in the case). Here, the best configurations were chosen according to the percentage of runs in which a data race was detected. The other noise

DOI: 10.1002/stvr

configurations were, in fact, capable of detecting an error in most of the test cases too, but in less test runs.

The graph shows that even when the test cases are very similar and contain the same type of concurrency errors, most of the configurations work only for some of the test cases. Of course, one can see that some of the configurations were more successful than the others. In general, configurations using the *sleep* and *busyWait* heuristics were the most successful ones. The most successful approach was to combine these heuristics with the *random-all*, *read/write*, or *pattern* heuristics.

A further analysis of the results has also shown that choosing the right combination of noise placement and noise seeding is important, but tweaking the values of noise frequency and strength may also significantly influence the results. Many configurations provided very different results when the values of frequency or strength were changed.

As for the newly proposed heuristics, configurations using the *pattern* heuristic proved to be very useful in most of the test cases (namely, the `t01`, `t03`, and `t06` test cases). On the other hand, the *inverseNoise* heuristic helped only a little and only when combined with the *random-all* heuristic. As for the heuristics tested for the first time in C programs, namely the *sharedVar-all* and *sharedVar-one* heuristics, these heuristics achieved good results for some test cases, but they were not so good overall compared to the other noise placement heuristics.

4.5. A Comparison of Noise Injection Techniques in Java

In this section, new experiments with noise injection techniques in Java are presented. Similarly to the the results for C presented above, description of the testing environment and test cases is given first. Subsequently, a summary of the obtained results is presented.

4.5.1. Testing Environment The code instrumentation and noise injection was done using the IBM ConTest framework [14] executed with plug-ins implementing the noise heuristics and collecting selected coverage information. Automatic test instrumentation, execution, and evaluation was orchestrated by the SearchBestie framework [33]. In contrast to the experiments with noise injection to C programs, the infrastructure collected not only information related to execution time and error manifestation but also coverage under two selected coverage metrics, namely, HBPair* and Avio*, which were already used in the previous comparisons mentioned above and which were chosen due to their very good ratio of providing satisfactory results in the experiments with saturation-based testing described in Section 3.4 and a relatively low overhead of measuring the achieved coverage.

In the comparison below, all the configurations previously described in the comparison of noise heuristics for testing C/C++ programs were used, together with several more configurations based on the coverage-based heuristic [36] introduced in Section 4.1. Hence, the following noise placement heuristics are considered: *random-all*, *sharedVar-all*, *sharedVar-one*, *pattern*, *read/write*, and the coverage-based *coverage* heuristic which is exclusive to the comparision of Java programs. The reason why the *coverage* heuristic is studied only for the Java programs is the fact that the ANaConDA framework, used in the experiments with C programs, is not currently able to provide any coverage information and thus cannot support any coverage-based heuristics. All the heuristics were parametrized by the frequency parameter set to 15 % or 30 %. Note that this is the first time the *read/write* and the newly proposed *pattern* heuristic are evaluated on Java test cases.

As for the noise seeding heuristics, the same heuristics as in the C comparison above are considered, namely, *yield*, *sleep*, *busyWait*, and the newly proposed *inverNoise* heuristics. Again, two levels of noise strength for each of the heuristics were used: 2 and 20 nanoseconds for the *sleep* and *busyWait* heuristics, 2 and 20 instructions for the *inverNoise* heuristic, and 10 and 100 executions of `yield()` for the *yield* heuristic. Finally, experiments with the configuration which injects no noise into the execution but which instruments the code and collects coverage information were evaluated as well (referred to as *nonoise* below). Recall, that execution of any injected code in fact influences performance and scheduling of threads.

The above described combinations of heuristics give 97 noise configurations (6×2 noise placement heuristics, 4×2 noise seeding heuristics, and 1 *nonoise* configuration). Similarly to

DOI: 10.1002/stvr

the comparison for C programs, each configuration was given a 20 minutes time slot to test the considered program.

The above described configurations of noise injection techniques were evaluated on 8 Java test cases based on 6 Java programs of various size. The *Airlines*, *Crawler*, and *FtpServer* test cases have already been introduced in Section 3.4.1. The *Animator* test case is based on a simple graphic application for algorithm animation called *XtangoAnimator*. The test case creates a window and draws a picture according to a given batch file. The test case consists of 31 classes and contains a data race that leads to `NullPointerException`.

The *Rover* test case is a Java version of the NASA Ames K9 Rover Executive [22]. The test case, consisting of 83 classes, executes a selected high-level plan or plans—programs written in a language that specifies actions and constraints on the movement, experimental apparatus, and other resources of the rover. The test case contains a deadlock and a data race in the testing environment during exchanging of two consecutive high-level plans. Both errors make the test hang. Similarly to the *Crawler* test case, the probability of spotting the errors is extremely low without the use of a noise injection.

The *Elevator* test case is a simple real-time discrete event simulator [60] which contains atomicity violation leading to `NullPointerException`. Elevators are modeled as individual threads that poll directives from a central control board. The communication is synchronized using locks. The used configuration simulates 4 elevators.

Moreover, to demonstrate that the testing environment also plays an important role in the testing process, two prominent test cases in which the probability of spotting the error is extremely low (namely, Rover and Crawler) were executed on two different hardware configurations (the results are then referred to as *Crawler2* and *Rover2*). The Airlines, Animator, Crawler, and Rover test cases were executed on Intel i5-2500 machines with 2GB memory running Linux with the 2.6.32 kernel and 64bit Sun (Oracle) JVM version 1.6. The Crawler2, Elevator, FtpServer, and Rover2 test cases were executed on Intel i7-3770K machines with 4GB memory running Linux with the 3.2.0 kernel and 64bit OpenJDK JVM version 1.6.

4.5.2. Experimental Results In this section, results comparing efficiency of the considered noise configurations from the most important point of view, namely their efficiency in error detection, are presented. Then, a short discussion of the results these heuristics achieved in terms of coverage is provided. Next, results achieved by the newly proposed heuristics are highlighted. And finally, the influence of the testing environment is discussed.

In a vast majority of the test cases, the error does not manifest during the 20 minutes long testing of non-instrumented code. Instrumentation of the test cases usually increases the probability to spot an error a bit because the instrumented code is executed in locations suitable for noise injection. In particular, the *nonoise* configuration was able to detect an error within the given time slot in two test cases, namely, the Airlines (the error manifested in 8 % of runs) and FTPServer (the error manifested in 66 % of runs).

The success of noise-based testing in detection of concurrency errors is summarized in Figure 3. The figure shows configurations grouped by the noise placement and noise seeding heuristics on the x-axis (the noise frequency and strength are represented by the different hatch of the bars). The y-axis shows the number of test cases (out of 8) in which the particular configuration was able to detect concurrency errors within the given time. In most cases, there were only a few configurations which were able to detect errors in the given time (ranging from 2 in the Elevator test case to 9 in the Crawler test case). In the Airlines and FTPServer test cases where the probability of spotting an error is much higher than in the other test cases, all of the noise configurations were able to detect the errors.

The figure shows that there is no silver bullet among the considered heuristics. Indeed, none of the columns reached value 8 which would mean that the heuristics worked for all the considered test cases. Moreover, one can clearly see that some of the configurations were mostly successful (mainly the configurations combining the *coverage* heuristic with the *sleep* and *busyWait* heuristics)

DOI: 10.1002/stvr

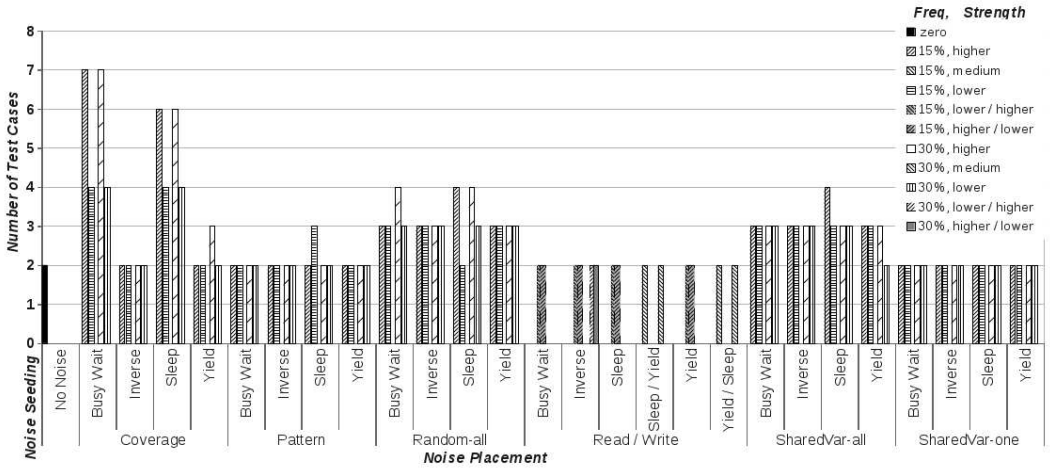

Figure 3. A comparison of noise configurations across all of the Java test cases

and some were successful only in the easy Airlines and FTPServer test cases (for instance, the *pattern* noise placement heuristic combined with most of the noise placement heuristics).

A further analysis of the results also shows that in most of the cases choosing the right combination of noise placement and noise seeding heuristics was far more important then tweaking the noise frequency and noise strength parameters. Many configurations provided similar results when any value of strength and frequency was used.

Overall, the results focused on the detection of concurrency errors show that noise-based testing is able to dramatically increase the probability of finding concurrency errors. It is enough to use any combination of noise injection heuristics in order to detect errors that do manifest during normal test executions even through only rarely (as can be seen from the Airlines and FTPServer test cases). Moreover, in the case of truly rarely manifesting concurrency errors which are hard to spot even during the noise-based testing, a careful choice of the combination of noise placement and noise seeding heuristics and their parameters is necessary.

As for the coverage obtained under considered coverage metrics, the results clearly show a positive impact of noise-based testing in comparison with the *nonoise* configuration. In some cases, a high achieved coverage correlated with a success in error detection (for instance, in the Elevator test case), sometimes this correlation could be identified only between the error detection ability and one of the coverage metrics (e.g., Avio* metric in the Airlines test case and HBPair* in the Rover2 test case), and sometimes there was no correlation between the error detection ability and any of the considered metrics (for instance, in the Animator and FTPServer test cases). Therefore, one cannot claim that there is in general a correlation between the ability to detect errors and to achieve a high Avio* or HBPair* coverage. However, a further analysis of the results indicates that it might be the case that if the error depends on a behavior reflected by the coverage metrics, the configurations which achieve a high concurrency coverage are able to detect the error (for instance, Avio* and the atomicity violation in the Airlines test case and HBPair* and the deadlock error in the Rover test case).

Next, we discuss efficiency of the newly proposed heuristics (namely, the *pattern* and *inverseNoise*) and the *read/write* heuristic. The newly proposed heuristics did not help much in detection of concurrency errors which was a bit surprising because the preliminary results obtained on the Rover test case (when the noise with frequency of 5 % only was injected and coverage data were not collected) the combination of the newly proposed heuristics achieved the highest

DOI: 10.1002/stvr

error detection improvement[†]. Nevertheless, the heuristics achieved good results in obtaining a high coverage in some cases (e.g., in the Airlines test case). On the contrary, the *read/write* heuristic achieved very good results in improving the ability to detect concurrency errors in the Airlines and FTPServer test cases. Errors in these test cases were found by all the considered noise configurations, but noise configurations with the *read/write* heuristic increased the percentage of the detected erroneous runs the most.

Finally, the influence of the testing environment described in Section 4.5.1 (in particular, the different hardware used) on programs under test was analyzed on the Crawler/Crawler2 and Rover/Rover2 test cases. In the Crawler/Crawler2 test cases, the results clearly show the influence of the environment. The error was detected by 9 noise configurations in the Crawler test case. In the Crawler2 test case, the number of successful configurations increased to 39 including all the 9 configurations which worked for the Crawler test case. Additionally, in the Crawler2 test case which was executed on a machine with more available cores as described above, the obtained results show higher numbers of achieved coverage and a higher error detection ratio (i.e. the number of executions in which a suitable configuration was able to detect the error). Conversely, in the Rover/Rover2 test cases, the influence of the environment was minimal. The same configurations were able to detect the error and the achieved coverage reached almost the same levels.

4.6. Specifics of Noise Implementation

Before the comparison of the results obtained for C and Java test cases is provided, differences in implementing the noise injection techniques for C/C++ and Java programs are discussed here. There are various ways to insert noise injection code into a program. The code might be inserted directly to the source code of the program, to its intermediate code (e.g., Java bytecode), or to the binary code. In general, inserting the code to the source code of the program have several disadvantages. It requires to have the source code of the program (and all of the libraries it uses), which might not always be available. It is also less precise as the compiler might, e.g., move the code elsewhere because of some optimizations. Therefore, the ANaConDA framework [21] used for the C/C++ programs and the IBM ConTest framework [14] used for the Java programs insert the noise injection code on the binary and bytecode levels, respectively. In this section, a short summary of the experiences with implementing the noise injection techniques on the binary level of C/C++ code and the Java bytecode level is presented.

Inserting some code to the bytecode of a program is not a big problem as the bytecode instructions are quite simple and JVM uses minimum optimizations complicating this task. However, inserting code to binaries of a C/C++ program is not such easy task. On the binary level there are used highly optimized instructions such as conditional and repeat instructions [20]. While the conditional instructions might not be executed when the control reaches them, the repeat instructions may be executed more than once as though they were placed in a loop. Moreover, the `rep`-prefixed instructions, designed for manipulating continuous sequences of memory locations (e.g., within string operations), are both conditional and repeat instructions since they may be executed a fixed number of times, until some condition is met, or sometimes not executed at all. When the binary code contains such instruction, one has to be sure that the noise is injected only when the instruction was really executed or every time the instruction was executed in a loop.

Distinguishing local and shared variables represents another problem. In Java, local data are stored on the current thread stack and possibly shared data are stored on the heap. Since there exist different instructions for accessing stack and heap, it is easy to distinguish accesses to the heap and apply noise only to them. On the binary level, local variables are on the stack too but the stack is just a reserved part of memory which might be accessed in the same way as the memory containing globally accessible variables. If noise injection before accesses to local variables is not desirable, one has to determine before each access whether the accessed variable is stored in memory containing the stack or not.

[†]A hypothesis to be tested in the future is that the positive impact of the new heuristics on error detection is reduced by the further noise associated with collecting coverage data.

DOI: 10.1002/stvr

Finally, in some cases (e.g., in the implementation of the access pattern detector for the *pattern* noise placement heuristic), tracking of method or function entry and exit events is necessary. Again, such events were fairly easily identified in Java bytecode but fairly difficult to detect on the binary level of C/C++ programs where returning from functions is often heavily optimized by the compiler, e.g., using jumps between functions with the effect of the control effectively returning from another function than the one that was called [20], etc.

To sum up, the implementation of the actual noise generators is of equal difficulty in C and Java. On the other hand, instrumentation and execution monitoring is much harder on the binary level as described above. Overcoming the obstacles of the binary level optimizations has a negative impact on the overhead of the ANaConDA framework for noise-based testing and dynamic analysis.

4.7. Comparison of Results Obtained for C and Java Test Cases

In this section, discovered commonalities and dissimilarities when analyzing the obtained results of experiments with C and Java programs are briefly described. Note that this comparison may be partially influenced by the used infrastructures for noise injection in C and Java which differ as highlighted in the previous section and the test cases which are also not directly comparable (the comparison studies simple C programs created by students which implement a solution for the same problem and Java programs of various size implementing different problems). Nevertheless, the findings presented here might still be of interest for the users of noise-based testing.

The Java experiments indicate that the success of noise-based testing depends mainly on carefully choosing the noise placement and noise seeding heuristics (tweaking the frequency and strength parameters did not improve the results much). On the contrary, the results for the C programs show that strength and frequency of noise are also very important in the considered test cases. Further analysis of the results achieved for the Java Airlines test case indicate that they share a few characteristics with the C programs. In particular, the same heuristics (including the new ones) provided good results, tweaking of frequency and strength did considerably affect the results, and stronger noise provided often good results. The Airlines test case is of similar size, contains a similar data-depending error, and the error manifestation ratio without any noise heuristics is also comparable.

In the considered C test cases, the results clearly show that a majority of noise configurations provided similar results across the four considered test cases. A configuration which provided good results in one test case was successful also in the other test cases and vice versa the configurations which provided poor improvement achieved poor results in all considered test cases. This is most probably caused by the similarity of the test cases. Indeed, the very similar results were also achieved for the Crawler/Crawler2 and Rover/Rover2 test cases in Java. The configurations which provided good results for the Crawler (Rover) test case were among the good ones even under the slightly different conditions represented by the Crawler2 (Rover2) test cases.

4.8. Hints for Noise-based Testing

The results presented above indicate that there is no single optimal noise configuration.The same noise setting may provide significantly different results for different test cases, testing goals, as well as testing environment. Moreover, using a wrong noise injection technique can in some cases even 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 random setting which selects noise heuristics and their parameters at random before each execution of the tested program. This setting often does not achieve the overall best results as mentioned above but it provides reasonably good results with a minimal effort. Further, if one has at least a suspicion that the program under test may contain a data-dependent error (such as a data race or an atomicity violation), based on the experience, using some of the heuristics focused on shared variables (or restricting the random choice of noise heuristics to those focusing on shared variables) might be a good idea.

If one has to set up the noise seeding and placement heuristics manually (i.e., there is no support for the random choice of noise heuristics in repeated test), based on the results, one can recommend using the *yield*, *synchYield*, *wait*, and *busyWait* heuristics, which often provided good results in

DOI: 10.1002/stvr

the experiments described above. The *yield* and *synchYield* heuristics have a smaller impact on the performance while still providing good improvement in some cases. The *wait* and *busyWait* heuristics cause a considerable performance degradation, but they can help to test even rarely executed synchronization scenarios. Further, the results indicate that using a low noise frequency (in particular, below 5 %) or using a high noise frequency (in particular, over 50 %) does not bring a higher probability of spotting an error or obtaining a higher coverage. On the contrary, a high noise frequency used with a demanding heuristic (e.g., *busyWait*) has a negative impact on the efficiency of the test.

All the considered advanced noise seeding heuristics (i.e., *timeoutTampering*, *haltOneThread*) including the newly proposed heuristics (i.e., *coverage*, *read/write*, and *pattern-based*) 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. If they do, the results indicate that the same heuristics might be providing good results even if the test is executed in a different environment. This is because the efficiency of these heuristics depends on appearance of certain code patterns in the program under test. Therefore, a simple static analysis of the program might help with the decision making (e.g., an analysis which detects appearance of `wait()` and `sleep()` could indicate that the *timeoutTampering* heuristic might provide good results). Next, the results also indicate that heuristics which put noise at carefully selected locations only provide better results than heuristics which simply put noise randomly or at too many locations.

To sum up, above a number of hints that may be useful when applying noise-based testing is provided. But, it is important to repeat that choosing a suitable noise configuration is a difficult task, and the hints need not work in all cases. Hence, the final advice is to—if possible—experiment with more different noise settings. Moreover, in the next section, an automated approach to this problem based on using search techniques for finding suitable noise settings is presented.

5. META-HEURISTIC SETTING OF TEST AND NOISE PARAMETERS

As discussed above, there is no silver bullet among the many existing noise injection heuristics. Different noise heuristics provide different results when used with different programs, compiled by different compilers, and executed in different run-time environments (especially the processor type and system load influence the efficiency of the technique quite significantly). Moreover, some configurations can actually decrease the probability of an error manifestation. This is helpful for run-time healing of errors [31] but highly undesirable when trying to detect the errors.

The number of possible noise settings is usually very large, and it is not easy to find a suitable setting for the given program, its environment, and the goal of detecting, reproducing, or suppressing a certain error. 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 *meta-heuristic search techniques* [55] can help.

A *genetic algorithm* [55] is one of popular metaheuristic algorithms. GA starts by creating 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 represented by the individual. The next generation of individuals is obtained by a stochastic recombination (called a *crossover*) and *mutation* of selected individuals. Individuals are selected from the previous generation according to the value of their fitness.

Here, discussion on how a GA can be used to search for suitable types of noise heuristics and their parameters is provided as originally proposed by Hrubá et al. [26] (with some more details [27]). First, the task is formalized as the *test and noise configuration search problem* (the TNCS problem). Then, it is shown how to represent instances of this problem for a GA, and discussed which objective functions may be useful as building blocks of fitness functions suitable in the given context. Then, discussion of the GA parameters influencing solving the TNCS problem is provided. Next, an instantiation of the framework by providing a concrete fitness function suitable especially (but, as these experiments show, not only) in the context of data race detection is discussed. Finally,

DOI: 10.1002/stvr

experimental evidence that using a GA can indeed provide significantly better results than random noise injection is provided.

5.1. Related Work

Most existing works in the area of search-based testing of concurrent programs focus on applying various metaheuristics to control the state space exploration within *guided model checking* (GMC) [23]. The intention 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. Hence, these works concentrate on *searching for a walk in a directed graph* representing the state space generated by a model checker where the walk starts in an initial state and ends in an error state. Various metaheuristics, including simulated annealing [12], the genetic algorithm [4, 23], the partial swarm optimization (PSO) [12], and the ant colony optimization (ACO) [2, 3], have successfully been used within GMC to find deadlocks and/or assertion violations in simple concurrent programs and protocols. An advantage of GMC is that the underlying model checking offers a well-defined state space and a high degree of systematic. The approach, however, inherits limitations of model checking in terms of scalability and cost of the environment modeling. Noise injection instead focus on testing which is able to handle much larger real programs but does not provide such precision as GMC.

Debugging concurrent programs using noise injection and GA [17] focuses on making a known error show up during repeated test executions. Within the approach, the debugging problem is translated into the *test data generation* problem [41] where the goal is to automatically select inputs of the test such that a chosen testing goal is achieved. In particular, the approach uses a GA to search in the set of possible noise configurations C defined as the powerset of the disjoint union of sets S_V, S_A, and S_L where S_V defines the noise applied to selected program variables, S_A defines the noise applied to selected accesses to shared memory, and S_L defines the noise applied to selected concurrency related events (lock operations, etc.). The noise is determined by the type of noise together with the strength of the noise. The paper presents two objective functions ($size$ representing the produced amount of noise and $entropy$ encoding the probability of an error manifestation under the given noise setting) and a fitness function computed as a weighted combination of the objective functions. The fitness function therefore prefers configurations which make the error manifest with a high probability using a minimal amount of noise. The technique is evaluated on a set of small (hundreds of lines) Java programs that contain known concurrency errors which manifest quite often when noise injection is used. The authors claim that their approach is able to minimize the number of locations where to put noise and to increase the probability of an error manifestation. However, statistical data supporting this claim are missing in the paper.

Compared to the described approach [17], the below presented approach [26] does not search for concrete locations which should be noised with particular noise. Instead, the approach searches for noise seeding and noise placement heuristics (or combinations of these heuristics) and their parameters which can provide good results for a particular test and environment. This allows to use a simpler representation of individuals and to support much larger test cases with plenty of possible locations to be noised. Moreover, the approach presented below consider fitness functions which allow to focus not only on debugging but also on testing. The approach also considers and reflects the non-deterministic behavior of concurrent software. In particular, each individual is evaluated by a set of experiments and reevaluation of already evaluated individuals is performed.

Another popular approach to find suitable test inputs (including inputs for the noise generator) is the Combinatorial Test Design (CTD) technique [45]. CTD is a systematic approach that generates a set of test inputs such that the set contains all intended combinations of inputs. However, such a set is usually huge, and therefore the technique often focuses on covering all combinations of pairs of inputs only—such an approach is called *pair-wise testing*. As said, the technique covers the input space uniformly which means that the technique does not recognize promising areas to which the search-based approach invests more effort.

The debugging problem is targeted in other papers using probabilistic [8] and machine learning[58] algorithms instead of metaheuristics. In the probabilistic approach [8], program

DOI: 10.1002/stvr

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 the machine learning approach [58], a machine learning feature selection algorithm is used to identify a subset of program locations where to inject noise by correlating the selection of noised program locations with error manifestation.

Finally, an application of a steepest ascending search algorithm in the context of noise-based testing of concurrent programs has been studied [33] as well. The experiments showed that the local search technique tends to get trapped in a local optimum and is not suitable for the given setting in most of the cases.

Recently, an application of metaheuristics—in particular, the genetic algorithm—to the problem of unit test generation has been presented [53]. The technique produces a test suite (i.e., a set of unit tests) for a chosen Java class. Each unit test consists of a prefix p initializing the class (usually, a constructor call), a set of method sequences m that are to be called (each sequence therefore represents a computational thread), and a schedule s that is enforced during an execution of the test using a deterministic scheduler. Typically, there are a lot of possible unit tests (i.e., triplets (p, m, s)). Most of them are pruned away using various constraints on the number of considered schedules, which is done mainly by concentrating on the synchronization points (concurrency-relevant actions) present in the code and by considering only a limited number of schedules consisting of a small number of synchronization points. The GA is used to produce method sequences m (i.e., sequential executions) that are able to reach and execute selected synchronization points. Finally, the sequences are combined into multi-threaded unit tests examining the chosen schedules.

The above idea of a fully automatic generation of new unit tests based on a suitable combination of selected sequential programs seems to be promising. The deterministic approach of building schedules [53] can be replaced by heuristic noise injection as well. However, as already mentioned in Section 2, deterministic testing brings important benefits over noise injection when scalability is not an issue, which is usually the case for unit testing.

5.2. The Test and Noise Configuration Search Problem

As already discussed, there are two main issues to be solved when using noise injection: in particular, determining which program locations should be noised and which noise should be used. To solve these issues, one can use some *noise placement* and *noise seeding* heuristics, but there are many of them, and so some choice must still be done. Moreover, most of the heuristics are adjustable by one or more parameters influencing their behavior and efficiency (e.g., noise seeding heuristics are often parameterized by their strength). Further, one can combine several noise placement and noise seeding techniques within one execution. Indeed, as discussed above, such a combination provides in many cases better results than using a single heuristic. Finally, it is usually the case that there exist multiple test cases for a given program that can also be parametric.

To reflect the above, the *test and noise configuration search problem* (the TNCS problem) [26] has been formulated as the problem of selecting test cases and their parameters together with types and parameters of noise placement and noise seeding heuristics suitable for a certain test objective.

Formally, let $Type_P$ be a set of available types of noise placement heuristics each of which is assumed 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, one 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. Such elements are denoted C or *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. And finally, let $TC = T \times C$ be the set of *test configurations*.

Now, the TNCS problem can be defined as searching for a test configuration from TC minimizing or maximizing some chosen objective functions. One can also consider the natural generalization of

DOI: 10.1002/stvr

the problem to searching for a *set* of test configurations from TC minimizing or maximizing some chosen objective functions.

5.3. *Objective Functions for the Context of the TNCS Problem*

Several objective functions that can be useful in various instances of the TNCS problem are discussed here. Typically, the functions are combined into a single fitness function as illustrated in Section 5.5.

First, an objective function that can often be found useful is to minimize 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 maximize 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 analyzer* and to maximize the number of warnings about dangerous behavior 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 precision of the chosen analyzer. A high ratio of false positives and/or negatives makes this objective function unreliable.

A further possibility is to use some suitable *coverage metric* allowing one to judge how much of the possible behavior of the program under test has been covered (and hence how likely it is that some undesired behavior was omitted) and to look for test configurations maximizing the obtained coverage. One can, for example, use the concurrency-related metrics based on dynamic analyses that are discussed in Section 3.2 and that measure how many internal states a certain dynamic analyzer has reached. Of course, one can also consider various other existing coverage metrics, such as the synchronization coverage [11] that are also mentioned in Section 3.2 and that measures how well the various synchronization mechanisms used in the program under test are tested (by measuring how many different scenarios of the use of the synchronization 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 the focus is on 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. It is assumed that all of them are used in each testing run, which is consistent with the 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 minimize/maximize the number of enabled heuristics, volume or frequency of noise to be injected, etc. Such objective functions are typically not sensible alone, but

DOI: 10.1002/stvr

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.

5.4. A Genetic Approach to the TNCS Problem

Now, a way how a GA can be used to solve the TNCS problem is described. The approach is presented on a concretization of the TNCS problem for the context of using the ConTest tool [14] (with some extensions) for noise-based testing. The concrete set of considered noise configurations is described first. Subsequently, it is presented how one can apply the GA in this concrete setting.

5.4.1. ConTest-based Noise Configurations Below, the original noise injection heuristics of ConTest (cf. Section 4) together with the coverage-based noise placement heuristics from Section 3.2 are considered. Hence, three noise placement heuristics are available: the *random* heuristic which picks program locations randomly, the *sharedVar* heuristic which focuses on accesses to shared variables, and the *coverage-based* heuristic which focuses on accesses near a previously detected thread context switch. The *sharedVar* heuristic has two parameters modifying its behavior with 5 valid combinations of its values. The *coverage-based* heuristic 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* heuristic is enabled by default when $noiseFreq > 0$. The *random* heuristic can be suppressed by one parameter of the *sharedVar* heuristic 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* heuristic randomly reduces the time-outs used in the program under test in calls of `sleep()` (to ensure that they are not used for synchronization).

5.4.2. Individuals, Their Encoding, and Genetic Operations on Them In order to utilize a GA to solve the TNCS problem with the considered set of noise configurations, the particular test configurations can play the role of *individuals*. The test configurations are encoded 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. Here, however, the noise configurations are targeted only, which form vectors of numbers in the range $(0, 0, 0, 0, 0, 0)$–$(1000, 5, 3, 6, 2, 2)$. 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. Entries whose values are ordered are called *ordinal entries* below.

The standard one-point, two-point, and uniform element-wise (any-point) *crossover operators* [55] are considered in the form they are implemented in the ECJ library [61]. *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 [55] (with the standard deviation set to 10 % of the possible range or minimal value 2). Finally,

DOI: 10.1002/stvr

standard proportional and tournament-based fitness selection operators [55] implemented in ECJ are considered.

5.4.3. Parameters of Genetic Algorithms and the TNCS Problem GAs are adjustable through a number of parameters influencing the efficiency of the search process. The way these parameters should be set to make the search process as efficient as possible depends on the considered problem. Therefore, setting of these GA parameters is described now. In particular, one has to consider 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. Another rising question is whether it is better to run a few big populations or instead more small populations in case the time for testing is limited.

Due to a high cost of evaluating each test configuration through multiple executions, all the experiments were conducted on one selected case study only. In particular, the *Crawler* test case introduced in Section 3.4.1 was considered. With the aim of observing as many behaviors differing in their various important concurrency-related aspects as possible, a fitness function was instantiated to maximize the obtained coverage under three different concurrency coverage metrics, namely *Synchro*, *Avio**, and *HBPair** discussed in Section 3.2. This way, three different aspects of concurrency behavior is covered: interleaving of accesses from different threads to shared memory locations via *Avio**, successful synchronization of program threads inducing a happens-before relation via *HBPair**, and information about whether the implemented synchronization does something helpful via *Synchro*. The value of the fitness function was computed as accumulated coverage obtained from five executions of *Crawler* with the same test and noise setting. All experiments were evaluated using three statistics: (1) the average fitness value in each population, (2) the best individual fitness in each population, and (3) the cumulative value of fitness from all already evaluated individuals. For brevity, below, the conclusions that were derived from the obtained results are presented only—more details about the actual experiments and their detailed analysis are available in the original technical report [27].

The experiments were divided into three series. In the first one, the focus was devoted to the *population size*, *crossover*, and *mutation* operators. The results indicate that a small or middle size of populations (20 or 40 individuals) are more suitable in this context since they could achieve better results in a shorter time than bigger populations (100 or 200 individuals). Further, it was concluded that uniform crossover achieves a better coverage than one-point or two-point crossover, and that higher values of mutation probability (0.5 and more) have a counter-productive impact on finding the best solution (they become essentially comparable with random searching). In order to obtain the best solution in the shortest time, the size of population 20, uniform crossover with probability 0.25 and mutation 0.01 were chosen for the rest of the experiments presented next.

In the second series of experiments, the influence of *elite* and *random individuals* in the population was studied. Results show that without elite and random individuals, the GA did not manage to find the best possible solution. Adding a small number of elite individuals (10 % of the population), improved the quality of the discovered solutions although the best solution was still not found. Adding a relatively high number of elite individuals (20 % of the population) allowed the GA to find the best solution, but on the other hand, the time needed to get it was quite significant, and the cumulative coverage in the population was getting worse. Adding random individuals to the population caused the results to be less stable, but overall, it had a positive effect on finding better solutions and on increasing the cumulative coverage (even in the case when there were no elite individuals). Hence, the best configuration for getting the best solution and the biggest cumulative coverage is to use a high number of elite and random individuals. However, this solution is time consuming. Therefore, as a compromise, it was decided to use a relatively small number of elite individuals (10 % of the population) and no random individuals in the population.

In the last set of experiments, the *selection operator* to be used was selected. In particular, the fitness proportional selection operator as well as the tournament of two and four individuals were considered. Moreover, all their possible combinations were considered as well. The different settings

DOI: 10.1002/stvr

did not lead to big differences in the obtained results, yet the combination of the tournament of four individuals with the fitness proportional selection operator seemed to be slightly winning.

To sum up, the final values of the parameters of the GA that are considered when evaluating the below discussed application of the TNCS problem for improving data race detection are the following: Size of population 20, a combination of the fitness proportional selection operators with the tournament of four individuals, the uniform crossover with a higher probability (0.25), a low mutation probability (0.01), and two elites (that is 10 % of the population).

5.5. A Concrete Application of the Proposed Approach

In this subsection, a concrete application of a GA in the process of noise-based testing of concurrent programs [26] is described. In particular, the TNCS problem was instantiated by providing a fitness function combining some of the discussed objective functions with the aim to be useful for improving data race detection. The focus is devoted to finding the best test configuration which is motivated by its possible use in subsequent repeated testing of the given application (e.g., within regression testing). The experimental results show that the described solution can indeed significantly help. Moreover, the experiments show that the approach helps not only in finding data races but also other kinds of concurrency-related errors. In addition, it turns out that when the entire random testing process with the testing process used for finding the best test configuration are compared, the latter achieves better results despite the used GA was not primarily designed for controlling the entire testing process—this is indeed an interesting challenge for the future.

5.5.1. A GoldiLocks-based Objective Function Based on the experience with different concurrency coverage metrics and dynamic error detectors, fitness function was build such that it maximizes the coverage obtained under the concurrency coverage metric *GoldiLock* [32] (cf. Section 3.2) based on the GoldiLocks algorithm [16], together with maximizing the number of actual warnings produced by this algorithm. The *GoldiLock* has been chosen as the basis of the fitness 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, as discussed earlier, the concurrency coverage metric *GoldiLock* has multiple further positive properties. In particular, the coverage under this metric usually grows smoothly (i.e., with a minimum of shoulders) and does not stabilize too early (i.e., before most behaviors relevant from the point of view of data race detection are examined). Further, based on the discussion presented in Section 5.3, an intention to minimize the execution time and to maximize the number of detected errors were considered as well.

In summary, the presented approach aims at (1) maximizing coverage under the concurrency coverage metric *GoldiLock*, (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.

As discussed in Section 3.2, the *GoldiLock* metric counts the encountered internal states of the GoldiLocks algorithm, here optimized by using the short circuit (SC) evaluation [16]. In particular, the coverage tasks of the *GoldiLockSC* metric are tuples $(pl, var, state, goldiLockSet)$ where pl identifies the program location at which some shared memory location var is accessed, $state \in \{SCT, SCL, LS, E\}$ denotes the internal state of the GoldiLocks algorithm with respect to the use of SC checks and $goldiLockSet$ represents the lockset computed by the GoldiLocks algorithm when in the LS state. The approach weights the coverage tasks of this metric according to their severity. Namely, 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 weight 1 was assigned. The SCL and LS states mean that the first check does not succeed, but it is possible to use further heuristic short circuit checks (SCL) or use the full algorithm (LS) to infer an object (or objects) that proves correctness of the access. Such tasks were assigned with weight 5. Finally, the E state means that the algorithm detected a data race and produced a warning. Such tasks were weighted with 10. The weighted coverage is denoted as $WGoldiLockSC$.

DOI: 10.1002/stvr

A GoldiLocks warning has the form of a tuple $(var, ploc_1, ploc_2)$ where var identifies a shared variable, and $ploc_1$ and $ploc_2$ 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 $ploc_2$ or var. The number of different warnings issued during the test execution is denoted by $GLwarn$, and it was given the weight of 1000.

Finally, as already mentioned, aim was also devoted to 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, the fitness function is then defined as follows (expecting the time to be measured in milliseconds): $(WGoldiLockSC + 1000 * GLwarn + 10000 * error)/time$.

5.5.2. Case Studies The above described approach was evaluated on five test cases containing concurrency-related errors (in particular, data races, atomicity violations, and/or deadlocks as described below). The test cases are listed in Table IV. 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. The considered case studies are not very large, which is *not* due to a bad scalability of the approach, but rather due to the large number of experiments needed to evaluate it, which in summary take a lot of time even on small benchmarks.

The *Airlines*, *Crawler*, and *FTPServer* test cases have already been mentioned in Section 3.4.1. The *Airlines* test case contains a high-level atomicity violation and has 3 parameters: the number of resellers (1–5), the number of customer sets (1–5), and the number of customers in each set (1–10). The *Crawler* test case has no parameters and contains a data race which leads to unhandled exceptions. The *FTPServer* test case has one parameter setting the number of clients connecting to the server. It contains several data races that can cause exceptions. The *Animator* test case is based on a simple graphic application for algorithm animation called *XtangoAnimator*, and the *Rover* test case is a Java version of the NASA Ames K9 Rover Executive. Both these test cases have been already introduced in Section 4.5.1.

The *Airlines* and *Animator* experiments were run on Intel Core2 6600 machines. The *FTPServer* test case was run on a machine with two Intel X5355 processors. The *Rover* test case was run on a machine with the Intel i5-2500 processor. Finally, the *Crawler* test case was run on two different machines to demonstrate that concurrency error detection is environment dependent. The genetic approach is, however, able to automatically find the best configuration for each environment. In particular, the *Crawler* test case was run on the Intel i5 661 processor and the *Crawler** test case on the machine with four AMD Opteron 8389 processors.

5.5.3. Experimental Results For the actual experiments, the infrastructure described in Section 5.4.2 and the setting of parameters of the GA inferred in Section 5.4.3 were used. Although this setting was inferred for a different fitness function while using sampled values of the *noiseFreq* parameter only, it represents a good option even for other experiments with the GA. Indeed, the fitness function used in Section 5.4.3 was designed to be rather general to cover a lot of different concurrent behaviors. Moreover, analysis of the correlation between the values of the fitness function of Section 5.4.3 and the $GoldiLocSCk$ metric used in the GoldiLocks-based fitness function on the performed experiments shows 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, the elite individuals were allowed to be re-evaluated in the following generations. This is motivated by the fact that a few executions of an individual (5 in this 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 heuristic is used [36]. 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.

The genetic approach was compared with the random approach to the choice of noise heuristics and their parameters. In the random approach, 2000 test and noise configurations were randomly

DOI: 10.1002/stvr

Table IV. All columns denoted Error and Time provide relative improvements obtained using the GA against the random approach (for the concrete meaning see the text)

Test case			Best individual					Search process		
Name	kLOC	Params	Gen.	Error		Time		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

selected and evaluated by the infrastructure in the same way the individuals in the genetic approach were evaluated. Table IV summarizes obtained results. The table is based on average results gathered 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.

Evaluation of the best individuals. The middle part of Table IV (*Best individual*) contains five columns which compare the best individual obtained by the GA and found by the random approach. The *Gen.* column contains the average number of generations (denoted as gen below) within the best individual according to the considered fitness function was discovered. The numbers indicate that the GA was able to find the best individual according to the considered fitness function within the first quarter of the considered generations. This motivates a possible future work on designing a suitable termination condition for the GA-based testing process.

The *Error* column of the *Best individual* section of Table IV compares the ability of the best individual to detect an error. The column contains two sub-columns. The values in the first sub-column are computed as the fraction of the average number of errors found by the best individual computed by the GA 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 values in the second sub-column are computed as the fraction of the average number of errors found by the best individual computed by the GA 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 (this situation is discussed in more detail below).

Similarly, the *Time* column of the *Best individual* section of Table IV compares average times needed to evaluate the best individual obtained by the GA and the best individual found by the random approach. Again, there are two sub-columns. The value in the first sub-column 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 value in the second sub-column 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 IV represent how many times the GA outperforms the random approach. In general, one can see that the best individual found by the GA has a higher probability to spot a concurrency error, and it also needs less time to do so. Even if one let the random approach to perform 2000 evaluations, the best individual found by the GA 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 GA was not able to detect an error and some of the best individuals found by the random

DOI: 10.1002/stvr

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 GA 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 future research, one may think of further adjusting the fitness function such that this phenomenon is suppressed.

An evaluation of the search process. The right part of Table IV (*Search process*) provides a different point of view on the presented results. In this case, the goal is not in finding just one best individual learned genetically or by random generation that is assumed to be subsequently used in debugging or regression testing. Instead, the focus is devoted to the results obtained during the search process itself. The GA is hence considered here to play a role of a heuristic 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 this case). Note, however, that despite this comparison is provided and the GA turns out to work well even in this comparison, the used GA was not primarily designed for controlling the entire testing process but for finding the best individual test only (a design of a GA designed with a stress on controlling the testing process remains an interesting challenge for the future).

This part of the table contains three columns comparing the genetic and random approaches wrt. their successes in finding errors and wrt. the time needed to perform the 2000 evaluations. The first sub-column of (*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 second sub-column of *Error* column compares the average number of errors detected by the GA and the random approach when the latter is provided with the same amount of time as the GA. Finally, the *Time* column compares the average total time needed by the random approach in 2000 evaluations and the average time needed by the GA. Again, values higher than 1 in all the columns represents how many times the GA outperforms the random approach.

The cumulative results presented in the *Error* column show that the GA mostly outperforms the random approach. The exceptions in the first sub-column of *Error* column reflect the already above mentioned preference of the execution time in the fitness function, which is further highlighted by the *Time* column. For instance, in the worst case (the *Crawler* test case), the GA is more than 3 times faster but in total discovers three times less errors. Conversely, in the best cases (the *Airlines* and *Rover*), the GA 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 GA needed just 10.5 hours), and the average time needed to evaluate 2000 random individuals of the biggest test case *FTPServer* took 101 hours (whereas the GA needed on average just 53 hours).

Overall, the results show that the GA outperforms the random approach. They also indicate that one should probably partially reconsider the 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 the described fitness function helps to improve the testing process even for test cases that do not contain a data race. This can be attributed to that it favors configurations within which the synchronization occurs more often and therefore is tested more. The results obtained from the experiments with the *Crawler* test case evaluated using two different hardware configurations indicate that the GA is able to reflect the environment and focus on the noise heuristics and their parameters which provide better results for the considered environment.

6. CONCLUSION AND FUTURE WORK

In this paper, noise-based testing that helps to examine different thread interleavings during testing and dynamic analysis of concurrent programs and hence increases chances of finding concurrency-related errors was presented. An overview of multiple results in the area of noise-based testing recently published by the authors was given, including novel coverage metrics suitable for

DOI: 10.1002/stvr

saturation-based testing and search-based testing of concurrent programs. Then, various existing and two new heuristics for the noise placement and noise seeding problems that play a crucial role in noise-based testing were presented. Results of some previously performed comparisons of noise injection techniques were summarized and used as a basis for performing a new, more thorough comparison of the most promising noise injection heuristics as well as the new noise heuristics proposed in this article. The heuristics were compared according to their ability to find concurrency errors, to increase concurrency coverage (namely, under the Avio* and HBPair* coverage metrics), and to cause an acceptable performance degradation.

The presented experimental results show that noise injection can indeed very significantly improve the testing process, but there is no silver bullet among the many noise injection techniques. Their performance depends on the test case, test goal, as well as test environment. Hence, for a new test case, experimenting with the various noise injection heuristics may be needed—or, as often done in industrial practice (e.g., within the industrial use of ConTest in IBM mentioned in Section 2), one can apply a randomly selected mix of the heuristics. In order to improve on this aspect of noise-injection based testing, an application of the genetic algorithm for choosing a suitable combination of noise placement and noise seeding heuristics (and of the values of their parameters) was presented as well. Experiments obtained with this approach showed that it can indeed significantly improve the noise-based testing process despite there is still a lot of space for further improvements (as mentioned below too).

Several promising directions for future work were envisaged: (1) One could think of new heuristics and approaches combining the simplicity of noise injection with the recent developments in the field of deterministic testing. For instance, one could use noise-based testing to roughly explore the behavior of the tested program and use deterministic testing to test only particular areas of the program behavior. (2) There is a significant space for developing better fitness functions and better algorithms for search-based testing with noise injection. The results presented in Section 5 demonstrated that putting too much stress on minimizing the overhead caused by noise injection may be counterproductive in some cases. In fact, the problem of choosing a suitable noise injection technique is of a rather multi-objective nature, and so employing multi-objective optimization algorithms might help here. (3) Collecting concurrency-related coverage from many executions produces a huge amount of data. Data-mining techniques could therefore be used to mine new helpful knowledge about the program under test from these data. (4) Noise injection is a lightweight testing approach that has a moderate impact on the performance of the test. Nevertheless, there is a simple possibility to further improve its performance by using partial instrumentation of the code. In this case, only selected parts of the code would be instrumented, and therefore affected by the noise. All parts of the code which would be known to be safe or to contain no concurrency related behavior could be omitted during the instrumentation. (5) Finally, there is still a lot of space for new combinations of static and dynamic analyses, further improving efficiency of the testing processes.

REFERENCES

1. Power Framework Delay Fuzzing. Online at: http://msdn.microsoft.com/en-us/library/hh454184(v=vs.85).aspx, April 2013.
2. Enrique Alba and Francisco Chicano. Finding Safety Errors with ACO. In *Proc. of GECCO'07*. ACM, 2007.
3. Enrique Alba and Francisco Chicano. Searching for Liveness Property Violations in Concurrent Systems with ACO. In *Proc. of GECCO'08*. ACM, 2008.
4. Enrique Alba, Francisco Chicano, Marco Ferreira, and Juan Gomez-Pulido. Finding Deadlocks in Large Concurrent Java Programs Using Genetic Algorithms. In *Proc. of GECCO'08*. ACM, 2008.
5. Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proc. of ASWEC'01*. IEEE Computer Society, 2001.
6. Cyrille Artho and Armin Biere. Applying Static Analysis to Large-Scale, Multi-Threaded Java Programs. In *Proc. of ASWEC '01*. IEEE Computer Society, 2001.
7. Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008.
8. Yosi Ben-Asher, Yaniv Eytani, Eitan Farchi, and Shmuel Ur. Noise Makers Need to Know Where to be Silent – Producing Schedules That Find Bugs. In *Proc. of ISOLA'06*. IEEE Computer Society, 2006.
9. Yosi Ben-Asher, Eitan Farchi, and Yaniv Eytani. Heuristics for Finding Concurrent Bugs. In *Proc. of IPDPS'03*. IEEE Computer Society, 2003.

DOI: 10.1002/stvr

10. Saddek Bensalem and Klaus Havelund. Dynamic Deadlock Analysis of Multi-threaded Programs. In *Proc. of HVC'05*. Springer-Verlag, 2006.
11. Arkady Bron, Eitan Farchi, Yonit Magid, Yarden Nir, and Shmuel Ur. Applications of Synchronization Coverage. In *Proc. of PPoPP'05*. ACM, 2005.
12. Francisco Chicano, Marco Ferreira, and Enrique Alba. Comparing Metaheuristic Algorithms for Error Detection in Java Programs. In *Proc. of SSBSE'11*. Springer-Verlag, 2011.
13. Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.
14. Orit Edelstein, Eitan Farchi, Evgeny Goldin, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Framework for Testing Multi-threaded Java Programs. *Concurrency and Computation: Practice and Experience*, 15(3-5), 2003.
15. Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby, and Shmuel Ur. Multithreaded Java Program Test Generation. *IBM Systems Journal*, 41, 2002.
16. Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. Goldilocks: A Race and Transaction-aware Java Runtime. In *Proc. of PLDI'07*. ACM, 2007.
17. Yaniv Eytani. Concurrent Java Test Generation as a Search Problem. *Electronic Notes in Theoretical Computer Science*, 144, 2006.
18. Yaniv Eytani and Timo Latvala. Explaining Intermittent Concurrent Bugs by Minimizing Scheduling Noise. In *Proc. of HVC'06*. Springer-Verlag, 2007.
19. Jan Fiedor, Vendula Hrubá, Bohuslav Křena, and Tomáš Vojnar. DA-BMC: A Tool Chain Combining Dynamic Analysis and Bounded Model Checking. In *Proc. of RV'11*. Springer-Verlag, 2012.
20. Jan Fiedor and Tomáš Vojnar. Noise-based Testing and Analysis of Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of PADTAD'12*. ACM, 2012.
21. Jan Fiedor and Tomáš Vojnar. ANaConDA: A Framework for Analysing Multi-threaded C/C++ Programs on the Binary Level. In *Proc. of RV'13*. Springer-Verlag, 2013.
22. Dimitra Giannakopoulou, Corina S. Pasareanu, Michael Lowry, and Rich Washington. Lifecycle Verification of the NASA Ames K9 Rover Executive. In *Proc. of ICAPS'05*. AAAI Press, 2005.
23. Patrice Godefroid and Sarfraz Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2), 2004.
24. Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing Concurrent Programs to Achieve High Synchronization Coverage. In *Proc. of ISSTA'12*. ACM, 2012.
25. David Hovemeyer and William Pugh. Finding Concurrency Bugs in Java. In *Proc. of PODC'04*, 2004.
26. Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, Shmuel Ur, and Tomáš Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. In *Proc. of SSBSE'12*. Springer-Verlag, 2012.
27. Vendula Hrubá, Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. Testing of Concurrent Programs Using Genetic Algorithms. Technical Report FIT-TR-2012-01, Faculty of Information Technology BUT, 2012.
28. Pallavi Joshi, Chang-Seo Park, Koushik Sen, and Mayur Naik. A Randomized Dynamic Program Analysis Technique for Detecting Real Deadlocks. In *Proc. of PLDI'09*. ACM, 2009.
29. Devin Kester, Martin Mwebesa, and Jeremy S. Bradbury. How Good is Static Analysis at Finding Concurrency Bugs? In *Proc. of SCAM '10*. IEEE Computer Society, 2010.
30. Bohuslav Křena, Zdeněk Letko, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Shmuel Ur, and Tomáš Vojnar. A Concurrency Testing Tool and its Plug-ins for Dynamic Analysis and Runtime Healing. In *Proc. of RV '09*. Springer Verlag, 2009.
31. Bohuslav Křena, Zdeněk Letko, Rachel Tzoref, Shmuel Ur, and Tomáš Vojnar. Healing Data Races On-the-fly. In *Proc. of PADTAD'07*. ACM, 2007.
32. Bohuslav Křena, Zdeněk Letko, and Tomáš Vojnar. Coverage Metrics for Saturation-based and Search-based Testing of Concurrent Software. In *Proc. of RV'11*. Springer-Verlag, 2012.
33. Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar, and Shmuel Ur. A Platform for Search-based Testing of Concurrent Software. In *Proc. of PADTAD'10*. ACM, 2010.
34. Zdeněk Letko. *Analysis and Testing of Concurrent Programs*. PhD thesis, Faculty of Information Technology BUT, 2012.
35. Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proc. of PADTAD'08*. ACM, 2008.
36. Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. Influence of Noise Injection Heuristics on Concurrency Coverage. In *Proc. of MEMICS'11*. Springer-Verlag, 2012.
37. Shan Lu, Weihang Jiang, and Yuanyuan Zhou. A study of interleaving coverage criteria. In *Proc. of ESEC-FSE'07*. ACM, 2007.
38. Shan Lu, Joseph Tucek, Feng Qin, and Yuanyuan Zhou. AVIO: Detecting Atomicity Violations via Access Interleaving Invariants. In *Proc. of ASPLOS'06*. ACM, 2006.
39. Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. In *Proc. of PLDI'05*. ACM, 2005.
40. Jeremy Manson, William Pugh, and Sarita V. Adve. The Java Memory Model. In *Proc. of POPL'05*. ACM, 2005.
41. Phil McMinn. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability*, 14(2), 2004.
42. S. Morasca and M. Pezzè. Using High-level Petri Nets for Testing Concurrent and Real-time Systems. In *Proc. of RTSTA'90*. Elsevier, 1990.
43. M. Musuvathi, S. Qadeer, and T. Ball. CHESS: A Systematic Testing Tool for Concurrent Software. Technical Report MSR-TR-2007-149, Microsoft Research, 2007.
44. Mayur Naik, Chang-Seo Park, Koushik Sen, and David Gay. Effective Static Deadlock Detection. In *Proc. of ICSE '09*. IEEE Computer Society, 2009.
45. Changhai Nie and Hareton Leung. A Survey of Combinatorial Testing. *ACM Comput. Surv.*, 43(2), February 2011.

DOI: 10.1002/stvr

46. Chang-Seo Park and Koushik Sen. Randomized Active Atomicity Violation Detection in Concurrent Programs. In *Proc. of SIGSOFT'08/FSE-16*. ACM, 2008.
47. Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.
48. Eli Pozniansky and Assaf Schuster. Efficient On-the-fly Data Race Detection in Multithreaded C++ Programs. In *Proc. of PPoPP'03*. ACM, 2003.
49. Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A Dynamic Data Race Detector for Multi-threaded Programs. In *Proc. of SOSP'97*. ACM, 1997.
50. Koushik Sen. Race Directed Random Testing of Concurrent Programs. In *Proc. of PLDI'08*. ACM, 2008.
51. Elena Sherman, Matthew B. Dwyer, and Sebastian Elbaum. Saturation-based Testing of Concurrent Programs. In *Proc. of ESEC/FSE'09*. ACM, 2009.
52. Javier Soriano, Miguel Jimenez, Jose M. Cantera, and Juan J. Hierro. Delivering Mobile Enterprise Services on Morfeo's MC Open Source Platform. In *Proc. of MDM'06*. IEEE Computer Society, 2006.
53. Sebastian Steenbuck and Gordon Fraser. Generating Unit Tests for Concurrent Classes. In *Proc. of ICST'13*. IEEE Computer Society, 2013.
54. Scott D. Stoller. Testing Concurrent Java Programs using Randomized Scheduling. *Electronic Notes in Theoretical Computer Science*, 70(2), 2002.
55. El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley Publishing, 2009.
56. Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural Testing of Concurrent Programs. *IEEE Trans. Softw. Eng.*, 18, March 1992.
57. Ehud Trainin, Yarden Nir-Buchbinder, Rachel Tzoref-Brill, Aviad Zlotnick, Shmuel Ur, and Eitan Farchi. Forcing Small Models of Conditions on Program Interleaving for Detection of Concurrent Bugs. In *Proc. of PADTAD'09*. ACM, 2009.
58. Rachel Tzoref, Shmuel Ur, and Elad Yom-Tov. Instrumenting Where It Hurts: An Automatic Concurrent Debugging Technique. In *Proc. of ISSTA'07*. ACM, 2007.
59. Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model Checking Programs. In *Proc. of ASE'00*, page 3. IEEE Computer Society, 2000.
60. Christoph von Praun and Thomas R. Gross. Object Race Detection. In *Proc. of OOPSLA'01*. ACM, 2001.
61. David White. Software Review: The ECJ Toolkit. *Genetic Programming and Evolvable Machines*, 13, 2012.
62. Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. Sound and Precise Analysis of Parallel Programs through Schedule Specialization. In *Proc. of PLDI'12*. ACM, 2012.
63. Cheer-Sun D. Yang, Amie L. Souter, and Lori L. Pollock. All-du-path Coverage for Parallel Programs. In *Proc. of ISSTA'98*. ACM, 1998.
64. Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A Coverage-driven Testing Tool for Multithreaded Programs. In *Proc. of OOPSLA'12*. ACM, 2012.

DOI: 10.1002/stvr