A Platform for Search-Based Testing of Concurrent Software

Bohuslav Křena, Zdeněk Letko, Tomáš Vojnar
Brno University of Technology
Božetěchova 2, Brno
CZ 612 66, Czech Republic
{krena, iletko, vojnar}@fit.vutbr.cz

Shmuel Ur
IBM, Haifa Research Lab
Haifa University Campus
Haifa, 31905, Israel
ur@il.ibm.com

ABSTRACT

The paper describes a generic, open-source infrastructure called SearchBestie (or S'Bestie for short) that we propose as a platform for experimenting with search-based techniques and for applying them in the area of software testing. Further, motivated by a lack of research on search-based testing targeted at identifying concurrency-related problems, we instantiate S'Bestie for search-based testing of concurrent programs using the IBM's concurrency testing infrastructure called ConTest. We demonstrate capabilities of S'Bestie on a series of experiments, which—despite we have just started our experiments with S'Bestie—also illustrate the fact that search-based testing can be quite useful in the context of testing concurrent programs.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Verification, Testing, Searching

1. INTRODUCTION

Concurrent, or multi-threaded, programming has become highly popular. New technologies such as multi-core processors have become widely available and cheap enough to be used in common computers. However, as concurrent programming is far more demanding, its increased use leads to a significantly increased number of bugs that appear in commercial software due to errors in synchronization of its concurrent threads. This stimulates a more intense research in the field of detecting and removing such bugs.

One of the most common approaches used for uncovering bugs in software is in general *testing*. However, finding good test cases that will exercise a large enough portion of the behaviour of programs is not easy. That is why, various approaches to automated test generation are sought, including the use of various advanced *search algorithms*, searching through the state space of possible test settings, yielding the so called *search-based testing* [23].

The problem of uncovering bugs in concurrency by testing is in particular difficult since the bugs may manifest only under very special scheduling circumstances. One of the ways to cope with this situation is to use techniques suitably influencing scheduling, such as *noise injection* [9], and then, of course, use repeated test execution. However, finding a good setting of the noise injection is not easy as well as it is not easy to see how much time to spend on a repeated execution of a single basic test and when to move to another one in order to maximize chances to find bugs within the available time. For solving these problems, one may again resort to search algorithms and obtain a *search-based concurrency testing* environment. Nevertheless, to the best of our knowledge, such a possibility has not attracted research attention yet.

Motivated by the above, we propose a new generic, open source platform intended to provide an environment for experimenting with search-based techniques as well as applying these methods in the area of testing. We call our platform **SearchBestie** (**S'Bestie** for short) as an abbreviation of Search-Based Testing Environment. Since we are in particular interested in testing of concurrent software, we instantiate the generic platform for concurrency testing, by linking it to the concurrency testing platform ConTest [9] from IBM. We illustrate on a series of experiments how our tool can be used and (despite our experiments are at the beginning) also that search-based testing combined with noise injection and repeated testing can be of interest in concurrency testing.

The rest of the paper is organized as follows. The next section contains a short overview of the state of the art in automatic and/or search-based testing and testing of concurrent software. Our generic search-based testing platform S'Bestie is described in Section 3. Section 4 describes an instantiation of S'Bestie for concurrency testing using ConTest. Section 5 illustrates on a series of experiments the possible uses of S'Bestie and provides an indication that search-based testing combined with noise injection is of interest for testing concurrent software. Section 6 concludes the paper and discusses multiple interesting directions for future work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PADTAD'10 July 12, Trento, Italy.
Copyright 2010 ACM 978-1-4503-0136-7/10/07 ...$5.00.

2. RELATED WORK

2.1 Search-based Testing

Automation of testing is highly demanded. However, automated test cases generation is challenging. For instance, the branch coverage problem involves finding a (partial) solution to the so called path sensitisation problem that is the problem of finding an input to drive the software down a chosen path. This problem is known to be undecidable [15]. Therefore, techniques that search for near optimal test sets in reasonable time are sought.

Search-based testing applies various search optimization techniques to the problem of test data generation. In principle, the test adequacy criterion is encoded as a fitness function, and an optimization technique uses this function as a guide to achieve a maximal test adequacy. The search-based approach is very generic because different fitness functions can be defined to capture different test objectives.

Search techniques can be divided into local, global and hybrid techniques. The *local search techniques* [21, 7] search for a better solution of the problem in the neighbourhood of the currently known best solution. Such techniques provide a relatively good performance but can get stuck in a local optimum. The *global search techniques* [8, 17, 3, 26, 21] search for a better state in the whole state space. Such methods overcome the problem of local optima but usually need many more states to be explored. Global search techniques are usually implemented using evolutionary algorithms as the so called *evolutionary testing*. The third class of techniques, referred as *hybrid search techniques* [15], tries to combine the previous two approaches to get rid of their disadvantages. These techniques usually provide a better performance than global search techniques and they are able to overcome the local optimum problem in some cases.

There have been published many works in the area of search-based testing in the past two decades [23]. The approach has been so-far successfully applied mainly to structural testing [23], with a particular focus on generating test inputs to achieve a maximal branch coverage, exception testing [26], mutation testing [17], stress testing [3], regression testing [21], interaction testing [7], temporal testing [3], and state machine testing [8]. According to our knowledge, search-based testing has not been applied to concurrent testing yet.

There have also appeared various tools supporting search-based testing, such as, e.g., EvoTest [14] and JAFF [2]. Compared with these tools, S'Bestie aims at proving a higher generality on one hand and at providing support for search-based testing in concurrency testing on the other hand.

2.2 Finding Bugs in Concurrent Software

Despite the immense research efforts devoted to the area of verification of concurrent software and the related area of detection of concurrency bugs, no silver bullet has been found in these areas. Among the most common approaches used for uncovering concurrency bugs there is *testing*, typically extended in some way to cope with the fact that concurrency bugs appear often only under very special scheduling circumstances. To increase chances of spotting a concurrency bug, various ways of *influencing the scheduling* are often used (apart from running the same basic test many times). An example of this approach is random or heuristical noise injection used in the ConTest tool [9], systematic forcing of some conditions on the program interleaving [27], or a systematic exploration of all schedules up to some number of context switches as used in the CHESS tool [24] (with the hope that a relatively low number of context switches is usually sufficient to uncover a bug). In this paper, we work towards combining ConTest and noise injection with search-based testing in order to ease the task of finding suitable parameters for noise injection (and also other parameters of testing, such as the criterion when to stop repeating a given basic test).

Another way to improve traditional testing in the area of concurrency is to try to extrapolate the behaviour seen within a testing run and to warn about a possible error even if such an error was not in fact seen in the testing run. Such approaches are called *dynamic analyses*. Many dynamic analyses have been proposed for detecting special classes of bugs, such as data races [10, 12], atomicity violations [22, 13], or deadlocks [1, 18]. These techniques may find more bugs than classical testing, but on the other hand, their computational complexity is usually higher and they can also usually produce false alarms. A use of some of the dynamic analyses within S'Bestie is an interesting subject for the future.

An alternative to testing and dynamic analyses is the use of various *static analyses* or *model checking*. Static analyses try to avoid execution of the given program or to execute it on a highly abstract level only. Various static analyses of concurrent software exist, including light-weight analyses that look for specific patterns in the code that might lead to a bug [16] or, e.g., various dataflow-based analyses that try to identify bugs like data races [11, 19] or deadlocks [29]. Model checking [6] (sometimes viewed as a heavy-weight static analysis too) tries to systematically analyze all possible interleavings of threads in a given program (the CHESS approach can, in fact, be seen as a form of bounded model checking). Light-weight static analyses may produce many false alarms, many static analyses may be specialised to a narrow class of bugs, and heavy-weight approaches may have troubles with scalability. There also exist approaches that combine static and dynamic analyses in an attempt to suppress their deficiencies [5, 20]. A use of static analyses to help search-based testing to focus on interesting parts of the tested code may be another interesting issue for future research.

3. THE ARCHITECTURE OF S'BESTIE

The S'Bestie infrastructure is divided into four cooperating modules called the *Manager*, *State space storage*, *Search*, and *Executor*. The roles and interactions of all these modules are briefly introduced in the following paragraph. Each module is then described in more detail together with some design and implementation notes in the following four subsections.

A general overview of the structure and functioning of the S'Bestie architecture is provided in Figure 1. The manager reads a configuration file and initializes other modules. Then, the manager enters a loop common for all search techniques. The manager asks a search engine to identify a state in the searched state space, which may be viewed as a test and its parameters, to be explored in the next step. The chosen state is then passed to the execution module that executes the appropriate test. Results of the test are collected, and an object encapsulating the results is passed back to

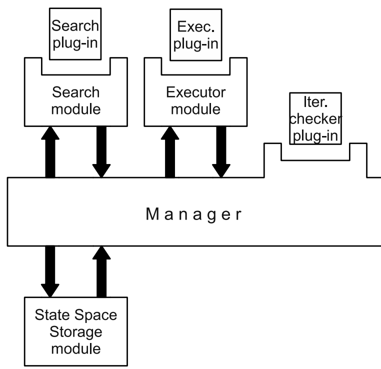

Figure 1: High-level architecture of S'Bestie

the search engine as a feedback, and then stored in the state space storage. Subsequently, a test whether pre-defined termination conditions have been fulfilled is performed. If not, the next iteration starts, and the manager asks the search engine to provide a next state of the state space to be explored. When the searching is over, the manager can analyze the obtained results or export them.

The architecture is meant to be very generic and therefore all modules consist of two parts: an interface that communicates with the rest of the infrastructure and pluggable engines that actually provide the functionality. Engines can implement the functionality on their own or can implement an interface to an external library or tools. Since engines for the same module share a common interface, they can be easily interchanged. This allows users to easily experiment with several different testing approaches. The generality of our architecture is also supported by the idea of building blocks that allow to combine several pluggable engines into more complicated engines.

3.1 The State Space Storage Module

The state space storage module encapsulates a model of the state space being searched and stores results obtained from the already explored states. The interface provided by this module allows other modules to get information concerning the state space and the results obtained so far.

The *state space* can have any number of dimensions, which may be of several different kinds. In particular, S'Bestie currently supports four types of dimensions: (a) boolean dimensions, (b) integer dimensions with given minimal and maximal values, (c) enumeration dimensions consisting of some number of values that can be strings, integers, and booleans, and, finally, (d) *test dimensions*, which are special dimensions containing an enumeration of tests that are used to explore particular states.[1] Each dimension contains information whether values in the dimension are considered to be ordinal or not. Each state space must contain exactly one test dimension. So, if the test dimension contains only one test, the whole state space is explored using this test. However, for complex programs, we expect dozens of tests. We will describe tests in more detail in the following subsection.

A *state* in a state space is then a vector of values from all dimensions (one value per dimension), having the semantics of a test and its parameters. Of course, there can be states that are not allowed or are not interesting from the testing point of view (e.g., some parameter does not influence the particular test or some combination of parameters needs not be valid for a particular test). Such combinations may be specified[2] and the search engine notified when it tries to explore an illegal state.

When a state is explored, the executor engine produces a result that is stored in the state space storage module. The storage of results has the form of a map that maps already explored states to the corresponding results. The module can be configured to store only one result for each state or to store a set of results. When a particular state is explored multiple times, the user can define whether all results are stored or only one value obtained by applying a combination operator on the old and new result value. The storage module currently supports three combination operators, returning the better (w.r.t. the result of a user defined fitness function), worse, or cumulated value (obtained by applying a user-provided function).

The *result* of a state exploration is a vector of values of various kinds and meanings. Each result contains at least two items in the vector that are introduced by S'Bestie. The first describes whether the test *passed*, *failed*, or was *interrupted*, and the second contains information on how long the exploration of the state took. Other values in the vector are considered to be problem-specific and left to the user to be defined. For instance, in our experiments, we use results containing two additional values where the first is a bit array representing coverage information and the second represents the number of errors detected during the test execution.

When dealing with search-based algorithms, a computation of the so called *fitness function* is important. Since there could be several different fitness functions to be used over a vector of values in a result, we decouple the evaluation of fitness functions from the computation of results. Therefore, the user can experiment with different fitness functions without a need to change anything within the results.

The state space storage module can automatically provide various statistical data such as the number of so far explored states, the number of obtained results or the number of results with some specific value, coordinates of notable states (e.g., providing the highest value of the fitness function), cumulative results cumulated over the whole state space or over a selected subset of states, etc. All data stored in the state space storage module can be exported and later imported. We also provide users with an interface that can be used for further analyses of the collected results or their export to proprietary formats. Using this feature, we have, e.g., implemented an algorithm that exports results into data sets which can then be used in GNUPlot[3] to visualize the state space. We have also implemented a simple correlation analysis among parameters and results. Another implemented algorithm allows state spaces to be exported into the Universal Java Matrix Library[4](UJMP), to present

[1] We use the name "test dimensions" as testing is the main application of S'Bestie for us. However, this does not restrict the use S'Bestie to testing only: the special "test" dimension can contain, in general, any functions to be run over particular states, leading to some evaluation of them, without any restriction of what is the real meaning of these functions.

[2] For example, by a user-defined predicate—this feature is not yet supported by S'Bestie.

[3] http://www.gnuplot.info/

[4] http://www.ujmp.org

results within a graphical interface of this library. UJMP allows to view two dimensional matrices with annotations, draw two dimensional graphs, and print some statistical information of the matrix or its submatrices. It also allows to make basic transformations of the matrix, e.g., normalization. From UJMP, the obtained results can further be exported in various formats including CSV.

3.2 The Executor Module

The Executor module controls exploration of a chosen state from the state space either by executing an external program (e.g., a test) or by evaluating a function encoded into a Java class. Again, the module can be divided into an interface communicating with the rest of the infrastructure and a pluggable engine that actually performs the computation. As was mentioned in the previous subsection, the engine is determined by the value of the test dimension of the particular state being explored.

To allow users to easily create more complicated engines, a few customizable building blocks are available. The building blocks are based on three kinds of engines: (1) A *task* performs some given work. Currently, tasks that execute processes or Java processes are available. (2) A *test* engine is very similar to a task engine with the exception that when a test is performed, it generates a result. Therefore, the test engine actually implements exploration of a given state. (3) An *iteration checker* engine takes as its parameter a generated result and decides whether a further exploration is needed or not.

Based on these basic engines, several more complex engines are available too: (1) A *timed test* encapsulates a given test engine and if an execution of the encapsulated test exceeds a given time bound, the execution is interrupted. (2) A *repeated test* adds an iteration checker engine to a test. The test is then executed in a loop until a termination condition is fulfilled. One can also choose whether results generated by the encapsulated test are stored in the state space storage module or whether the resulting (best, worst, cumulated) result is stored there only. (3) A *composed test* allows for adding additional tasks into the encapsulated test. One task is optionally executed before performing the test and the second one is optionally executed after the test.

Since executing hundreds of tests is time demanding, we also implemented a simple plug-in that imports data from a previously explored state space. Afterwards, when the plug-in is asked to perform a test, a result obtained within the imported exploration is immediately returned. This way, the experiments can be performed quite fast and the user can analyze results instantaneously as far as the configuration of the state space is the same and the plug-in is not asked to return more results than are available.

3.3 The Search Module

The search module consists of an interface which is used to cooperate with the rest of the infrastructure and a search algorithm implemented as a plug-in which actually does the search. A search module is called to provide the next state that should be explored. A search engine can access the state space storage module via an appropriate interface in order to get information concerning the state space (dimensions, results, statistics, etc.).

To prove the concept, we implemented three search algorithms: a sequential search that explores the whole state space in a sequential manner, a random search that explores randomly chosen (not yet explored) states, and two versions of the well-known hill climbing (greedy) search algorithm. One version of the hill climbing considers neighbouring states within a selected dimension only while the other considers neighbourhood over all dimensions.

3.4 The Manager Module

The manager module is a service module that takes care of the initialization and finalization of S'Bestie, provides supporting tools for other modules (e.g., the logging subsystem), and during the exploration loop controls the flow of data among modules. The manager module also provides a simple interface for a communication with the user.

S'Bestie takes as input a few basic parameters (specifying, e.g., whether to import previous results or not, etc.) and an XML configuration file. The file contains a configuration for all modules and is divided into 3 parts (a configuration of the search module, the executor module, and the state space storage module). The main part of the configuration file is devoted to a definition of the dimensions of the state space to explore.

The algorithm of the state space exploration in S'Bestie was already mentioned in the beginning of this section. Let us, however, describe it in more detail now:

1. Optionally, an initialisation task is performed before the actual exploration begins. This task can be used to prepare tests, e.g., compile and/or instrument the code.

2. The search module is asked to identify a state to be explored. Each state has the test dimension that identifies the test engine to be used for an exploration of the state.

3. The chosen test engine is initialized and executed, taking as a parameter the state. The test generates a result.

4. The result generated by the test engine is passed to the search module as a feedback for the search engine and to the state space storage module to be stored.

5. Finally, the result is passed to the iteration checker engine that decides whether a termination condition is satisfied. If not, the execution continues by Step 2. If the termination condition is satisfied, the execution continues by the next step.

6. Optionally, a finalization task is performed after the exploration. This task can be used to finalize tests, e.g., do some post-processing of the results, remove temporary files, etc.

There are multiple ways how the exploration loop may end. The exploration may be finished if a global timeout expires, the user makes the manager stop the exploration, the search engine does not provide any state to be explored, or the iteration checker executed at the end of each iteration detects that some pre-defined termination condition has been fulfilled. When the exploration loop is finished, the explored state space is exported to a file for further analysis. During the exploration, another output file is being produced. This file contains a report concerning the sequence of explored states and the corresponding achieved results

(which may itself be interesting when exploring, e.g., how a cumulated coverage was increasing during the exploration).

When experimenting with different heuristics, one sometimes needs to perform multiple state space explorations and to compute average values of the obtained results in order to decrease influence of the nondeterminism present in the used approach (e.g., when the starting point of the hill climbing algorithm is chosen randomly). Therefore, our manager module allows to set a number of times the process of exploration is performed and to collect important statistical data from such executions. Of course, it is also possible to subsequently analyze each exploration separately using logs and exported data.

4. SEARCH-BASED CONCURRENCY TESTING

The task of search algorithms in search-based testing is to generate test data that provide the best result according to some *measure* used to evaluate the success of the testing process. The commonly used measures have often the form of some sort of *coverage*. For instance, in structural search-based testing, the measure is code or branch coverage. The generated test data usually include a test setting (i.e., various parameters of the test) and sometimes also input data of the program under test.

We take this general idea and apply it to testing of concurrent software in the following way. We extend the test data generated by the search algorithm to contain also a configuration of the ConTest infrastructure that we use to facilitate concurrency testing. Since we focus on testing concurrency, we use fitness functions taking into account the achieved success in testing of concurrency (based on various coverage measures that ConTest provides to us). Therefore, we are able to guide search algorithms to focus on such test data (and hence ConTest configurations) that test concurrency better than other test data.

In the following, we shortly discuss the role of test repetition in concurrency testing. Then, the ConTest infrastructure is described, and concurrency related coverage measures that ConTest is able to produce are introduced. Finally, we describe how we incorporate ConTest into S'Bestie.

4.1 Test Repetition in Concurrency Testing

Testing is usually based on the idea that a test is executed and results (or partial results) are compared with the expected results. If the results obtained from the test correspond to the expected results, the test passes. If not, the test fails. Concurrency introduces non-determinism into the execution, and one typically has to check that a program produces the same results for the same inputs regardless of which of the many legal orders of operations within the program was used. Therefore, testing of concurrent software is known to be very difficult since a test that has already passed in many executions may suddenly fail just because the order of operations during the execution differs from all previous executions. Concurrency testing tools, like the one used by us, therefore focus on observing different legal orderings of program operations during multiple executions of the same test with an intention to find an ordering that makes the test fail. To sum up, *it is reasonable to execute a single basic test with the same inputs multiple times*.

A question is how many times the test needs to be executed. Our experiences show that the *number of previously unseen orderings decreases with the number of executions*, which happens even when heuristics such as those implemented in ConTest—trying to force different interleavings in different executions (as described in the following subsection)—are used. As explained below, we take these observations into account when instantiation S'Bestie for concurrency testing built on top of ConTest.

4.2 ConTest for Java Concurrency Testing

ConTest [9] is an advanced tool for testing, debugging, and measuring test coverage for concurrent Java programs. Its main goal is to expose concurrency-related bugs in parallel and distributed programs, using random noise injection. ConTest instruments the bytecode—either off-line or at runtime during class load—and injects calls to ConTest runtime functions at selected places. These functions sometimes try to cause a thread switch or a delay (generally referred to as *noise*). The selected places are those whose relative order among the threads can impact the result; such as entrances and exits from synchronised blocks, accesses to shared variables, and calls to various synchronization primitives. Context switches and delays are attempted by calling methods such as `yield()`, `sleep()`, or `wait()`. The decisions are random so different interleavings are attempted at each run, which increases the probability that a concurrency bug will manifest. Heuristics are used to try to reveal typical bugs. No false alarms are caused by ConTest because all interleavings that occur with ConTest are legal as far as the JVM rules are concerned. ConTest itself does not know that an error occurred. This is left to the user or the test framework to discern, exactly as they do without ConTest.

ConTest provides a possibility to measure several coverage measures during execution. Besides typical measures related to code coverage (method and basic block coverage), ConTest is able to produce also three concurrency related coverage measures: shared variable coverage, coverage of concurrent pairs of events, and synchronization coverage.

The *shared variable coverage* measure [4] considers as the so called coverable tasks (i.e., units of the coverage criterion) variables that might be shared. A task is covered within an execution if an instance of the corresponding variable has been accessed by two or more different threads during the execution. This information is important for system designers but does not provide much indication concerning how well the program is tested.

In the case of *coverage of concurrent pairs of events* [4], each coverable task is composed of a pair of program locations that are assumed to be encountered consecutively in a run and a third field 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 concurrency coverage model provides two important pieces of information: (a) Tasks containing the $false$ value provide code coverage information and (b) tasks containing the *true* value provide concurrency coverage information about context switches that occur at some places.

The *synchronization coverage* model [27] focuses directly on the synchronization primitives used, and, in particular, on whether they do something "interesting". The coverage model consists of tasks that describe all possible "interesting" behaviours of selected synchronization primitives. For

instance, in the case of a synchronized block (defined using the Java keyword `synchronized`), the related tasks are: *synchronization visited*, *synchronization blocking*, and *synchronization blocked*. The synchronization visited task is basically just a code coverage task. The other two are reported when there is an actual contention between synchronized blocks—when a thread $t1$ reached a synchronized block A and stopped because another thread $t2$ was inside a block B synchronized on the same lock. In this case, block A is reported as blocked, and block B as blocking (both, in addition, as visited). The synchronization primitives for which synchronization coverage is currently handled by ConTest are `synchronized` blocks and calls of `Object.wait()`.

4.3 Concurrency Testing in S'Bestie

We have combined ConTest with S'Bestie in order to allow search-based testing of concurrent software. We use concurrency coverage measures described in the previous subsection for detecting whether there is some progress when testing with some test data and also as inputs for computing fitness functions describing how well the program was exercised. To allow all this, we implemented the following plug-ins of S'Bestie.

First, we have implemented a task plug-in that executes the ConTest's off-line instrumentation on a selected code. We use the off-line instrumentation because it allows us to collect a list of coverable tasks for particular coverage models. This task is usually executed by the manager module before the main exploration loop is executed.

We have also implemented a specialised ConTest test plug-in which: (1) produces a configuration of ConTest for a particular run, based on values of the dimensions of a particular state to be explored, (2) executes a test in a separate Java process using ConTest with the given settings, (3) produces a result which may include some of the ConTest's measures of concurrent code coverage (depending on the user's choice). The achieved coverage is encoded in the form of a binary vector with an entry for each coverable task, which can easily be used for further manipulations, including the computation of fitness functions or the computation of cumulated results. In addition, we combined this plug-in with unhandled exception detection. The result generated by the ConTest plug-in therefore contains concurrency coverage information together with a number of detected unhandled exceptions.

Finally, we have used the repeated test building block from the executor module to implement an iterative checker for a repeated application of the same basic ConTest test from above. The iterative checker computes a difference between the last cumulated result and the cumulated result obtained some pre-defined number of iterations before. If the difference falls below a pre-defined threshold, the iteration is finished. The threshold can be set absolutely or relatively to the number of already covered tasks.

Features that were described above allow us to apply structural search-based black-box testing if standard code coverage measures are used, and what is more important, they also provide us with a platform for applying search techniques on concurrent software.

5. EXPERIMENTS

In this section, we provide an illustration of experiments that can easily be performed with S'Bestie. We focus on experiments from the area of concurrency testing and we try to not only demonstrate capabilities of S'Bestie but also to pinpoint various specifics of concurrency testing that can be of interest for future research on search-based testing of concurrent programs.

Program	Classes	kLOC	SynchroC	kCPairsC
Airlines	2	0.1	6	0.8
WebCrawler	19	1.2	47	15.6
FTPServer	120	12	22	180.4

Table 1: Programs used for experiments

5.1 Experimental Setup

For our experiments, we used three concurrent programs containing concurrency bugs. Table 1 provides basic data concerning all three examples: the number of classes, the number of lines of code (in thousands), the number of different tasks of synchronization coverage (*synchroCoverage*) that were in total covered by our tests, and the number of tasks of coverage of concurrent pairs of events (*concurrentPairsCoverage*) that were in total covered within our tests (in thousands). We use the numbers of tasks presented in Table 1 as the numbers of coverable tasks in the following since the exact number of coverable tasks cannot be easily determined.

The first program, called `Airlines`, is a simple, artificial program that demonstrates a bug caused by a missing synchronization of an operation that decrements a shared integer variable. When the program is started, it creates a flight with a predefined capacity (set to 150 in our case). Then, a predefined number of threads is executed each modeling a ticket seller. Each ticket seller decrements the free capacity of the flight by one until there is no seat available in the flight. Most ticket sellers use a correct synchronization protocol and a few do not. Finally, a number of sold tickets is compared with the flight capacity, and if they do not match, an exception is thrown. This exception can be used to track whether an error occurred during a particular execution.

The two other programs are real life case studies (see, e.g., for more detail [20]). `WebCrawler` is a part of an older version of a major IBM production software. The crawler creates a set of threads waiting for a connection. If a connection simulated by a testing environment is established, a worker thread serves it. There is a bug in a method that is called when the crawler shuts down. The bug causes an exception that can be tracked. The bug is so tricky that it reveals very rarely (approximately once in 10 thousand of executions).

The second real-life example is a development version of an open-source `FTPServer` produced by Apache. The server creates a new worker thread for each new incoming connection to serve it. The version of the server we used contains several data races that can cause an exception. Since the exception is thrown within the worker thread, more than one exception can arise during one execution. We set a timeout to the server so it shuts down after a certain amount of time. This is a different approach to the two previous examples where the amount of work that the threads had to process was constant.

The state spaces we dealt with in our experiments contain 7 dimensions and 2200 or 1760 states as can be seen in Ta-

Dimension	Type	Ord	Num.items
CTNoiseFreqency	enum of int	Y	11
CTNoiseType	enum of str	N	5
CTHaltThread	boolean	Y	2
CTTimeoutTamper	boolean	Y	2
CTSharedVarNoise	boolean	Y	2
NumThreads	enum of int	Y	5 (4)
Test	test	N	1

Table 2: State spaces used in experiments

ble 2. Let us shortly describe all the dimensions. The dimensions starting with CT are parameters of ConTest. The *CTNoiseFrequency* setting tells ConTest how often some noise should be caused. ConTest allows the frequency to be set by a number between 0 (no noise at all) and 1000 (noise all the time). Since dealing with this full range of values would lead to a huge state space, we decided to restrict this dimension to contain only 11 values (0, 100, ..., 900, 1000). The *CTNoiseType* setting tells ConTest what kind of noise generator to use. There are 5 possible values (*yields*, *sleeps*, *waits*, *synchYields*, and *mixed*, which each time randomly chooses one of the four previous ones). The remaining three parameters *CTHaltThread*, *CTTimeoutTamper*, and *CTSharedVarNoise* enable (or disable) three heuristics that ConTest uses to identify places where to put noise. The *NumThreads* dimension is not related to ConTest, it is a parameter of the tested applications—namely, it specifies how many competing threads to execute (or how many clients may connect to the FTPServer concurrently). The values of this setting are 2, 4, 8, 16, 32 and 2, 4, 8, 16 in the case of the FTPServer (giving 1760 states for this program). The *Test* dimension contains tests that can be executed. Hence our setup contains only one test for each program, the size of this dimension is 1.

Experiments with the first two programs were performed on Intel Core 2 Duo E8400 with 2 GB RAM and experiments with the third one on 2xIntel Xeon X5355 with 64 GB RAM. The amount of memory present on the testing machines was not limiting for us because our infrastructure used on average 60 MB of RAM to store the whole state spaces.

5.2 Exploring Selected States

As was already pinpointed in Section 4.1, concurrency testing differs from the classic testing mainly in the fact that a repeated execution of the same test can produce different results. As our initial experiment, we use the ability of S'Bestie to easily explore different chosen states a given number of times and to store the (accumulated) results as well as the history of their values to demonstrate this fact and to explore it in more detail. To do so, we direct S'Bestie to repeatedly execute the same test for a predefined number of times (set to 100) and collect changes in concurrency coverage after the particular executions.

Figures 2 and 3 show three different curves representing cumulative values of *concurrentPairsCoverage* that we obtained with 3 different configurations of ConTest in the Airlines and FTPServer examples. One configuration executes the test without ConTest noise injection[5], another with ConTest noise enabled (*CTNoiseFrequency* set to 600, *CTNoiseType* set to *mixed*), and yet another with ConTest noise enabled (600, *mixed*) and the *CTSharedVariable* heuristic enabled. The cumulative value of *concurrentPairsCoverage* allows us to see the progress of testing because it increases each time a test captures a behaviour that has not been seen before.

The graphs show that some configurations of ConTest can significantly increase the efficiency of testing while others can actually decrease it because the noise is caused in unsuitable places or there is too much noise put in too many locations (which is the case of our experiment). Note that the influence of the different heuristics is different in the two figures which is an illustration of a more general fact that each program is specific and needs a different setting of ConTest in order to achieve a better coverage, which gives space for search-based techniques to identify such settings.

As can be seen, the difference between consecutively obtained values of cumulative coverage usually decreases with the number of iterations. Another interesting observation that can be seen especially from the graph in Figure 2 is that at some points, the cumulative coverage makes shoulders (i.e., for a certain number of executions, the coverage remains constant and then starts growing again). Look, for instance, at samples 18 to 27 of the case when ConTest noise is disabled. The existence of shoulders is a sensitive issue when using the iteration checker described in Section 4.3. This checker stops iterating when the difference between n last cumulative results falls below a predefined threshold. If possible (e.g., based on experience from previous tests), the parameter n should be estimated with respect to the size of the shoulders. If n is chosen to be lower than the size of a typical shoulder plus one, the iteration checker may prematurely stop the iteration on some shoulder. On the other hand, setting n to a too high value may cause the iteration checker to iterate too many times with no effect.

Note also that when a test which is insensitive of concurrency is repeatedly executed, the cumulative value of *concurrentPairsCoverage* remains constant all the time. This can be used to identify such tests and to stop their repeated execution.

In the case of *synchroCoverage*, the number of coverable tasks is very low in our experiments as shown in Table 1. Therefore, the cumulative coverage based on *synchroCoverage* contains more shoulders, and the differences between configurations of ConTest are relatively small as can be seen in Figure 5.

5.3 Analysing Entire State Spaces

S'Bestie can also be easily used to explore an entire state space and collect the results for a further analysis—such an analysis can be useful, e.g., for choosing suitable test parameters for repeated testing, for testing similar applications, or when collected from a subspace of a larger state space, for driving further testing of the given application.

To illustrate various possibilities of using data collected by S'Bestie from some state space, we first show how the dependence of the cumulative value of *concurrentPairsCoverage* on test configurations can be analysed. We choose the WebCrawler case study for this purpose. The results obtained

[5]Note that ConTest routines collecting coverage information already represent some noise injected into the execution and therefore the concurrency coverage with really no noise may slightly differ.

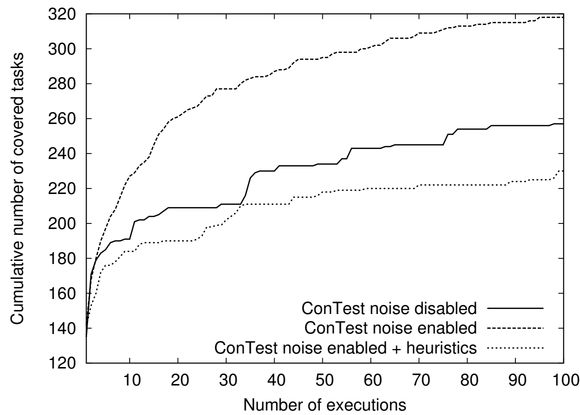

Figure 2: Influence of noise in the Airlines example

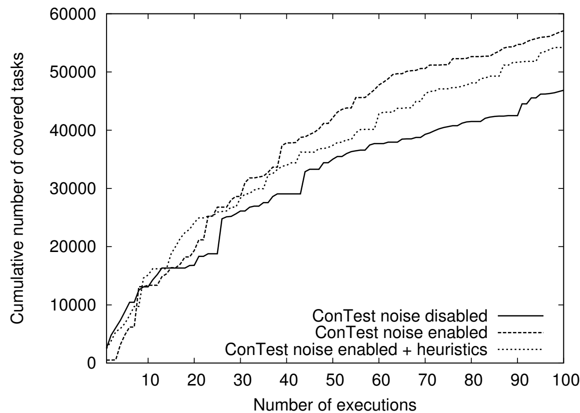

Figure 3: Influence of noise in the FTPServer case study

from the WebCrawler case study can be arranged as shown in Figure 4. The y-axis represents cumulative values of the coverage while the x-axis represents the used test configuration. The x-axis, in fact, comprises the 6-dimensional state space described in Table 2, which is collapsed as follows. By (manually) analysing the data obtained from S'Bestie, we have discovered that the parameter mostly influencing the coverage in this case is *CTSharedVarNoise*. *CTSharedVarNoise* is therefore used as the most significant parameter for ordering of configurations—the left half of the picture corresponds to the points in the state space where *CTSharedVarNoise* is disabled while the right half to the points where *CTSharedVarNoise* is enabled. The second most influencing parameter is *CTNoiseType*. The 1st and the 6th *"hill"* from the left in the picture corresponds to the *mixed* noise, the 2nd and the 7th correspond to the *sleep* noise, the 3rd and the 8th to the *synchYields* noise, the 4th and the 9th to the *waits* noise, and the 5th and the 10t to the *yields* noise. The last parameter that has a visible influence on coverage is *CTNoiseFrequency*—the frequency ranges from 0 % for the left part of the hills to 100 % for their right part (with 10% steps). When the noise frequency is set to zero, the noise type has no influence (one can see that each hill starts with a dip). Other parameters (i.e., *NumThreads*,

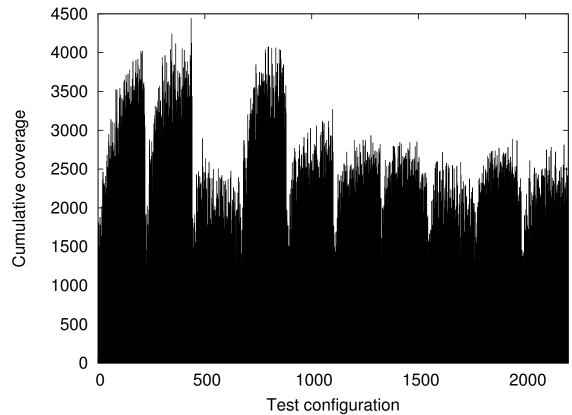

Figure 4: Dependence of cumulative coverage on test configurations in WebCrawler

CTTimeoutTamper, and *CTHaltThread*) have no visible influence on the coverage.

Based on the above results, one can easily deduce how to set parameters of ConTest in order to achieve good values of cumulative coverage for applications similar to WebCrawler: (1) disable *CTSharedVarNoise*, (2) choose the mixed, sleeps, or waits type of noise, (3) generate noise with high frequency, (4) select other parameters arbitrarily.

5.4 Exploring Already Explored State Spaces

S'Bestie can also be used for a repeated exploration of already explored state spaces based on the results stored from a previous exploration. This functionality can be useful for speeding up experiments with various searching strategies since the actual exploration of particular states is no more needed—we will get back to this in the next subsection.

The possibility to explore already explored state spaces can further be used, for instance, for finding good ways of traversing the given state space with respect to various criteria, taking into account the knowledge of results for all other states (which cannot be done within a regular search which knows about the so far explored states only). The discovered strategy can then be used, e.g., for repeated tests, for testing similar applications, or as an etalon for comparing with various studied search strategies.

We illustrate the above possibility on trying to identify a good sequence of states whose cumulated exploration covers as many concurrency coverage tasks as possible in our case studies. We use the following algorithm implemented on top of S'Bestie to find such a sequence of states. The algorithm takes the single state that covered the most tasks and computes cumulated values of this state when combined with all other states. The tuple that provides the best cumulated value is passed to the next iteration where a triple providing the best cumulated value is computed and so on. Our results show that by combining of a very small number of ConTest configurations, we are able to cover most of the program behavior.

Figure 5 shows the evolution of the *synchroCoverage* measure within 900 executions of the `WebCrawler` case study when a sequence of ConTest configurations identified by the above described algorithm is used compared to a random sequence of configurations. It demonstrates the situation

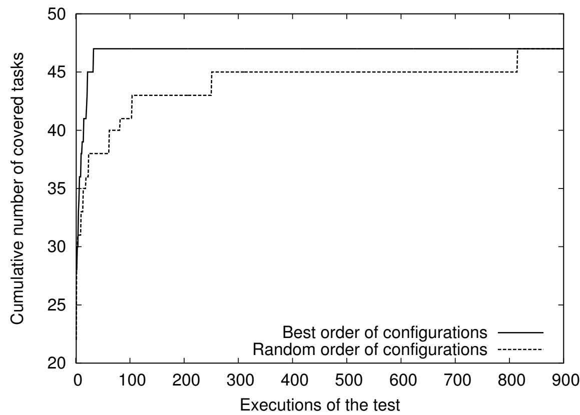

Figure 5: The influence of choosing different sequences of states to test

	CPairsC		SynchroC	
Program	Best	Rand.	Best	Rand.
Airlines	16	563	1	1
WebCrawler	598	2200	2	74
FTPServer	1126	1760	2	1690

Table 3: The number of iterations needed to reach the 100% coverage of coverable tasks through testing a sequence of states

captured in Table 3, namely, that using the sequence of configurations obtained by the algorithm described in the previous paragraph, we obtain the 100% coverage of coverable tasks within the first 35 executions: 20 executions of the first ConTest configuration and the rest with the second ConTest configuration—hence, the 100% coverage is obtained within 2 first configurations. A randomly chosen sequence of ConTest configurations reached the 100% coverage in 817 executions using 74 ConTest configurations. Table 3 shows the results for *concurrentPairsCoverage* and for our other case studies too. It can be seen that in general it is much easier to reach the 100% coverage of *synchroCoverage* than *concurrentPairsCoverage*. One can also see that a very small number of suitably chosen ConTest configurations can cover all interesting behavior of the program.

5.5 Using a Local Search Approach

One of the main goals of S'Bestie is to support applications of or experiments with different search techniques. In this section, we demonstrate the ability of S'Bestie to be used to compare two search heuristics. Despite the results we obtain from the experiments are preliminary (and the main goal was to demonstrate that S'Bestie can be used to perform this kind of experiments), the results also indicate that there is a real potential in using search-based techniques in testing of concurrent software.

In our experiment, we compare a simple hill climbing algorithm with a random search algorithm described in Section 3.3. We accelerated these experiments using the executor plug-in that imports a previously explored state space. As mentioned already in the previous subsection, this solution allows us to compare these two searching approaches because both of them explore the same state space. This cannot be achieved by executing real tests containing concurrency because of the nondeterminism in execution of concurrent software. We have also made the manager module to perform multiple state space explorations using the same heuristics in a loop (with 100 iterations) in order to get average values as was described in Section 3.4.

	CPairsC			SynchroC		
Program	Iter	Hill	Rand	Iter	Hill	Rand
Airlines	344	25%	17%	326	100%	100%
WebCrawler	366	36%	16%	371	14%	16%
FTPServer	277	11%	22%	301	29%	21%

Table 4: Results of searching for the best configuration

We choose to search for a suitable configuration of ConTest that provides the best coverage. To achieve this goal, we set the fitness function used by the hill climbing algorithm to represent a cumulated number of tasks covered in the particular state of the state space. The testing scenario was as follows: First, we identified the maximal value that is possible to reach (the best state in the state space). Then, we ran the hill climbing algorithm 100 times and got the average number of steps that hill climbing needed to reach the optimum (or local optimum). After each exploration, we also checked whether the algorithm reached the optimum. Finally, we ran the random search algorithm also 100 times and counted the number of times the algorithm found the optimum.

Table 4 summarizes our results. Column *Iter* shows how many states were explored on average by our hill climbing algorithm until the optimum was reached and two other columns show the percentage in which the algorithms found the optimum. Results for both considered coverage measures are provided. It can be seen that in 3 cases, the hill climbing algorithm beats the random approach, and in 2 cases, it does not. In the case of the `Airlines` example and the *synchroCoverage* measure, many states represent the optimum and therefore both algorithms reached 100%. The number of iterations in the case of `FTPServer` is lower because the state space itself is smaller (1760 states), and hence the hill climbing algorithm did not generate so many neighbours in each step.

6. CONCLUSIONS AND FUTURE WORK

We have described S'Bestie, a generic, open-source framework that we have proposed and implemented for experimenting with search-based techniques and, in particular, their applications in software testing. Moreover, we have instantiated S'Bestie for use in the area of concurrency testing by linking it with the ConTest infrastructure. We have illustrated capabilities of our framework on a number of experiments, which we have chosen such that they also illustrate that using search-based techniques in the context of concurrency testing is a promising direction.

As for the future work, there may be identified two main directions. The first is to improve our platform and the second is to focus on developing new heuristics and techniques for efficient testing of concurrency in complex software.

Regarding the first direction of improving the S'Bestie infrastructure, the possible steps may include:

- Implementing interfaces between S'Bestie and chosen libraries providing local and global search algorithms. This should provide interested users with a wide range of algorithms to use and show them how to implement an interface between S'Bestie and their favourite algorithms.
- Adding support for unit testing and tools that are used for unit testing such as, for instance, TestNG and JUnit. The intention is to provide a simple import procedure that would take existing tests of an application and process them such that they could easily be used in S'Bestie.
- Adding a support for dealing with state spaces that do not fit into the main memory and also for distributed exploration of the state spaces with the aim of obtaining a robust and scalable testing tool.
- Improving user interfaces of S'Bestie and allowing users to analyze and visualize interesting parameters

Concerning the second work direction of developing new heuristics and techniques for efficient testing of concurrency in complex software, the following interesting issues may be considered:

- ConTest provides, besides the already described features, also a listeners architecture [25] that allows one to incorporate algorithms for dynamic analysis into the exploration process. Combining dynamic analyses with search-based testing is an interesting direction that could allow one to further increase the number of uncovered bugs.
- Another promising application of our infrastructure could be in searching where to put noise. As was described in [28], the problem of putting noise at the right place in order to locate a concurrency bug can be very hard. We can create a dimension containing all program locations where it can be interesting to put some noise (identified, e.g., by some static analysis) and let a search algorithm to find such a combination of locations where to put noise such that a bug is revealed.
- In general, our infrastructure currently does black-box testing of concurrent software[6]. We would like to incorporate into the testing process also some static analyses that could be used to pinpoint interesting places in the program to focus on.
- Finally, our experiences show that S'Bestie is able to produce a huge number of results which might be interesting to analyze using data mining algorithms. Therefore, we plan to interconnect our state space storage module with a database and use data mining algorithms to identify interesting knowledge out of the results.

Acknowledgment. This work was supported by the Czech Science Foundation (project no. P103/10/0306 and doctoral project no. 102/09/H042), the Czech Ministry of Education (projects COST OC10009 and MSM 0021630528), and the internal BUT FIT grant FIT-10-1.

[6]ConTest heuristics are, however, based on a detection of specific code constructions.

7. REFERENCES

[1] R. Agarwal and S. D. Stoller. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. In *PADTAD '06*, pages 51–60, New York, NY, USA, 2006. ACM Press.

[2] A. Arcuri. Evolutionary repair of faulty software. Technical Report CSR-09-02, University of Birmingham, 2009.

[3] L. C. Briand, Y. Labiche, and M. Shousha. Stress testing real-time systems with genetic algorithms. In *GECCO '05*, pages 1021–1028, New York, NY, USA, 2005. ACM.

[4] A. Bron, E. Farchi, Y. Magid, Y. Nir, and S. Ur. Applications of synchronization coverage. In *PPoPP '05*, pages 206–212, New York, NY, USA, 2005. ACM.

[5] J. Chen and S. MacDonald. Towards a better collaboration of static and dynamic analyses for testing concurrent programs. In *PADTAD '08*, pages 1–9, New York, NY, USA, 2008. ACM.

[6] E. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

[7] M. B. Cohen, M. B. Dwyer, and J. Shi. Constructing interaction test suites for highly-configurable systems in the presence of constraints: A greedy approach. *IEEE Trans. Software Eng.*, 34(5):633–650, 2008.

[8] K. Derderian, R. M. Hierons, M. Harman, and Q. Guo. Automated unique input output sequence generation for conformance testing of fsms. *Comput. J.*, 49(3):331–344, 2006.

[9] O. Edelstein, E. Farchi, E. Goldin, Y. Nir, G. Ratsaby, and S. Ur. Framework for testing multi-threaded java programs. *Concurrency and Computation: Practice and Experience*, 15(3-5):485–499, 2003.

[10] T. Elmas, S. Qadeer, and S. Tasiran. Goldilocks: a race and transaction-aware java runtime. In *PLDI '07*, pages 245–255, New York, NY, USA, 2007. ACM Press.

[11] D. Engler and K. Ashcraft. Racerx: effective, static detection of race conditions and deadlocks. *SIGOPS '03*, 37(5):237–252, 2003.

[12] C. Flanagan and S. N. Freund. Fasttrack: efficient and precise dynamic race detection. In *PLDI '09*, pages 121–133, New York, NY, USA, 2009. ACM.

[13] C. Flanagan, S. N. Freund, and J. Yi. Velodrome: a sound and complete dynamic atomicity checker for multithreaded programs. *SIGPLAN '08*, 43(6):293–303, 2008.

[14] H. Gross, P. M. Kruse, J. Wegener, and T. Vos. Evolutionary white-box software test with the evotest framework: A progress report. In *ICSTW '09*, pages 111–120, Washington, DC, USA, 2009. IEEE Computer Society.

[15] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 99(RapidPosts):226–247, 2009.

[16] D. Hovemeyer and W. Pugh. Finding concurrency bugs in java. In *PODC '04*, July 2004.

[17] Y. Jia and M. Harman. Constructing subtle faults using higher order mutation testing. *Source Code Analysis and Manipulation, IEEE International Workshop on*, 0:249–258, 2008.

[18] P. Joshi, C.-S. Park, K. Sen, and M. Naik. A randomized dynamic program analysis technique for detecting real deadlocks. In *PLDI '09*, pages 110–120, New York, NY, USA, 2009. ACM.

[19] V. Kahlon, Y. Yang, S. Sankaranarayanan, and A. Gupta. Fast and accurate static data-race detection for concurrent programs. In *CAV*, pages 226–239, 2007.

[20] B. Křena, Z. Letko, Y. Nir-Buchbinder, R. Tzoref-Brill, S. Ur, and T. Vojnar. A concurrency testing tool and its plug-ins for dynamic analysis and runtime healing. In *RV*, pages 101–114, 2009.

[21] Z. Li, M. Harman, and R. M. Hierons. Search algorithms for regression test case prioritization. *IEEE Trans. Softw. Eng.*, 33(4):225–237, 2007.

[22] S. Lu, J. Tucek, F. Qin, and Y. Zhou. Avio: detecting atomicity violations via access interleaving invariants. In *ASPLOS-XII*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[23] P. McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, 2004.

[24] M. Musuvathi, S. Qadeer, T. Ball, G. Basler, P. A. Nainar, and I. Neamtiu. Finding and reproducing heisenbugs in concurrent programs. In *OSDI*, pages 267–280. USENIX Association, 2008.

[25] Y. Nir-Buchbinder and S. Ur. Contest listeners: a concurrency-oriented infrastructure for java test and heal tools. In *SOQUA '07*, pages 9–16, New York, NY, USA, 2007. ACM.

[26] N. Tracey, J. Clark, K. Mander, and J. McDermid. Automated test-data generation for exception conditions. *Softw. Pract. Exper.*, 30(1):61–79, 2000.

[27] E. Trainin, Y. Nir-Buchbinder, R. Tzoref-Brill, A. Zlotnick, S. Ur, and E. Farchi. Forcing small models of conditions on program interleaving for detection of concurrent bugs. In *PADTAD '09*, pages 1–6, New York, NY, USA, 2009. ACM.

[28] R. Tzoref, S. Ur, and E. Yom-Tov. Instrumenting where it hurts: an automatic concurrent debugging technique. In *ISSTA '07*, pages 27–38, New York, NY, USA, 2007. ACM.

[29] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP '05*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.