

Guided test for detecting concurrency errors

Neha Rungta
Department of Computer Science
Brigham Young University
Provo, UT 84601
neha@cs.byu.edu

Eric G. Mercer
Department of Computer Science
Brigham Young University
Provo, UT 84601
eric.mercer@byu.edu

ABSTRACT

Mainstream programming is migrating to concurrent architectures to improve performance and facilitate more complex computation. The state of the art analysis tools for detecting concurrency errors such as deadlocks and race conditions are imprecise, generate a large number of false error reports, and require manual verification of each error report. In this paper we present a guided test approach to help automate the verification of reported errors in multi-threaded Java programs. The input to the guided test is a small sequence of program locations (partial error trace) manually generated from the potential errors reported by the imprecise static analysis techniques. The guided test dynamically controls thread schedules during program execution in an effort to drive the program through the partial error trace and elicit a concurrency error. The scheduling decisions are made based on a two tier ranking system that first considers the portion of the error trace already observed and the perceived closeness to the next location in the error trace. The error traces generated by the technique are real and require no manual verification. We show the effectiveness of our approach in a set of benchmarked concurrency programs, and we automatically verify the existence of race conditions and deadlocks in the JDK 1.4 concurrent libraries as reported from the Jlint static analysis tool.

1. INTRODUCTION

The ubiquity of multi-core Intel and AMD processors is prompting a shift in the programming paradigm from inherently sequential programs to concurrent programs to better utilize the computation power of the processors. The shift from sequential programs to concurrent programs is accompanied with significant challenges. Although parallel programming is well studied in academia, research, and a few specialized problem domains, it is not a paradigm commonly known in mainstream programming. As a result, there are few, if any, tools available to programmers to help them test and analyze concurrent programs for correctness.

State of the art analysis tools [15, 12, 19, 1], for detecting concurrency errors are imprecise and generate a large number of false error reports. Static analysis techniques ignore data values of program variables and, thus, cannot prune infeasible execution paths during analysis. The static analysis tools look for either suspicious patterns [19, 1], rely on annotations [29, 15], or perform a control flow analysis of the system to check whether lock-acquisition locations in the program result in a cyclic dependency to indicate possible deadlocks [12, 32]. Since static analysis techniques ignore the data values of program variables and scheduling choices they can analyze large multi-threaded programs; however, the techniques cannot state the feasibility of the reported errors. As a result, the analysis is not sound and reports errors that *may* exist in the program and leaves it to the programmer to manually verify the errors by reasoning about input values, thread schedules, and control flow needed to manifest the concurrency error in an actual execution of the program. Relying on imprecise and unsound results that require manual verification is not feasible in mainstream software development because of complexity and cost associated with such an activity.

On the other end of the spectrum, model checking is a precise, sound, and complete analysis technique that reports only feasible errors. Model checking, [18, 31], is a formal verification approach that explicitly enumerates all possible behaviors of the program to check the presence and absence of errors. The possible behaviors of the system are collectively known as the state space of the program. Model checking techniques provide a concrete trace to an error; however, the complexity of concurrent systems leads to an exponential growth of the state space. This state space explosion has prevented the use of model checking in mainstream test frameworks.

In order to mitigate the state space explosion problem, directed model checking focuses its efforts in searching areas of the state space where an error is more likely to exist. It assumes the existence of the error. Directed model checking uses heuristic values and path-cost to rank the states in order of interest in a priority queue, [10, 17, 23]. The states estimated to be closer to an error location are explored before others. Directed model checking is effective in generating short counter-examples in models where depth-first and random search fail [24]. We have established in prior research on distance estimate heuristics [23, 22] for concurrent C programs that directed model checking can

quickly localize errors if the guidance algorithm has a notion of where the error is manifested in the program text. In large programs, however, the size of the search frontier grows very rapidly causing an explosion in the number of states saved in the priority queue which proves to be the bottleneck in the success of directed model checking.

In this paper we present a guided test technique, to automatically verify deadlocks and race conditions from error reports generated by imprecise static analysis techniques. The intuition is to use the imprecise analysis techniques to create an initial set of candidate error traces, and then use a precise technique, guided test, to verify the error traces and produce concrete program executions manifesting the real errors. Our solution would thus present to the developer a fully verified set of real deadlocks or race conditions with concrete error traces. The solution requires a reduced, and hopefully easier, manual effort in the verification process. The error traces generated by the technique are real and require no further verification; however, if the technique does not find an error we cannot prove the absence of the error; thus, our technique is sound in error detection but not complete.

The actual guidance is performed in a two step process. At points of non-determinism arising due to thread scheduling the states are first ranked using a new meta-heuristic. The meta-heuristic guides the test based on the portion of the error trace (sequence of locations) already observed. Intuitively, states that have observed a greater number of locations from the sequence are ranked as more *interesting* compared to the states that have observed fewer locations from the sequence. In the case where no new sequence locations are observed the guidance strategy uses a secondary heuristic to guide the search toward the next location in the sequence.

To test the validity of our guided test solution in aiding the process to automate the verification of deadlocks and race conditions, we present an empirical study in several Java programs collected from academic publications, IBM, classical concurrency problems, and the JDK 1.4 concurrent libraries. We show that guided test is extremely effective in localizing errors when given a few key locations in a potential error trace. The main contributions of this work are as follows:

- A guided test technique that automatically verifies deadlocks and race-conditions from a small sequence of program locations generated from potential errors reported by imprecise static analysis techniques.
- A guidance strategy using a two-tier ranking scheme where the states are ranked based on parts of the partial error trace already observed and a secondary heuristic value to guide the test toward the next location in the trace.
- An empirical study performed on multi-threaded Java programs (benchmarks and real examples) that demonstrates guided test to be more effective in defect detection when compared to a randomized depth-first search.

The primary limitations of this work are: (1) The technique does not consider any non-determinism arising due to data input; and, thus, it requires a closed-system and (2) Is effective only if the error specified by the partial trace exists in the program. We believe these are reasonable limitations since most classical black-box testing techniques require the tester to develop a closed system and proving the correctness of large multi-threaded systems is most often intractable which makes quick defect detection in such programs very appealing.

2. MOTIVATING EXAMPLE

We demonstrate with the example in Figure 1 the motivation for the guided testing approach presented in this paper. Figure 1 is a portion of a program using the JDK 1.4 public library. The `raceCondition` class in Figure 1(a) initializes two `AbstractList` data structures, l_1 and l_2 , using the synchronized `Vector` sub-class. Two threads of type `AThread`, t_0 and t_1 , are initialized such that both threads can concurrently access and modify the data structures, l_1 and l_2 . Finally, `main` invokes the `run` function of Figure 1(b) on the two threads. The threads go through a sequence of events, including operations on l_1 and l_2 in Figure 1(b). Specifically, an `add` operation is performed on list l_2 when a certain condition is satisfied that is later followed by an operation that checks whether l_1 equals l_2 . The `add` operation in the `Vector` class, Figure 1(c), first acquires a lock on its own `Vector` instance and then adds the input element to it. The `equals` function in the same class, however, acquires the lock on its own instance and invokes the `equals` function of its parent class which is `AbstractList` shown in Figure 1(d).

The question for the user is this, “Is there a concurrency error in this program?” A static analysis tool, Jlint [1], when run on the program in Figure 1, issues a series of warnings about potential concurrency errors in the JDK library. Jlint looks for suspicious patterns and detects cyclic lock dependencies to report potential deadlocks and race-conditions in Java programs. The Jlint warnings for the `equals` function in the `AbstractList` class in Figure 1(d) are on the Iterator operations (lines 8–12, and 15). The warnings state that the Iterator operations are not synchronized. As the program uses a synchronized `Vector` sub-class of the `AbstractList` the user may be tempted to believe that the warnings, in the current closed system, are spurious. A careful evaluation of the function, however, shows that the `equals` function in `Vector` Figure 1(c) when called from Figure 1(b) acquires a lock on l_1 , before calling the `equals` function in `AbstractList` but does not acquire a lock on the input list, l_2 , even though l_2 is also an instance of the synchronized `Vector` class. While a particular thread is iterating over the elements of l_2 , another thread can call the `add` function shown in Figure 1(b) that lead to a race-condition.

There is a need for a technique that can specifically test whether the race-condition specified with a partial error trace has a corresponding concrete error trace. For the example in Figure 1, we would like to ask, is it possible for one thread to call $l_2.add$ while another thread is iterating over the elements of l_2 in the `equals` function? The current testing techniques and tools do not allow us pose such a question. The technique presented in this paper allows us to guide the execution of a program along a partial error

```

1: class raceCondition{
2: ...
3: public static void main(){
4:   AbstractList l1 := new Vector();
5:   AbstractList l2 := new Vector();
6:   AThread t0 = new AThread(l1, l2);
7:   AThread t1 = new AThread(l1, l2);
8:   t0.start(); t1.start();
9:   ...
10: }
11: ...
12: }

```

(a)

```

1: class AThread extends Thread{
2:   AbstractList l1;
3:   AbstractList l2;
4:   AThread(AbstractList l1, AbstractList l2){
5:     this.l1 := l1; this.l2 := l2;
6:   }
7:   public void run(){
8:     ...
9:     if some_condition then
10:       l2.add(some_object);
11:     ...
12:     l1.equals(l2);
13:     ...
14:   }
15: }

```

(b)

```

1: class Vector extends AbstractList{
2: ...
3:   public synchronized boolean equals(Object o){
4:     super.equals(o);
5:   }
6: ...
7:   public synchronized boolean add(Object o){
8:     modCount ++;
9:     ensureCapacityHelper(elementCount + 1);
10:    elementData[elementCount ++] = o;
11:    return true;
12:  }
13: ...
14: }

```

(c)

```

1: class AbstractList implements List{
2: ...
3:   public boolean equals(Object o){
4:     if o == this then
5:       return true;
6:     if ¬(o instanceof List) then
7:       return false;
8:     ListIterator e1 := listIterator();
9:     ListIterator e2 := (List o).listIterator();
10:    while e1.hasNext() and e2.hasNext() do
11:      Object o1 := e1.next();
12:      Object o2 := e2.next();
13:      if ¬(o1 == null ? o2 == null : o1.equals(o2)) then
14:        return false;
15:    return ¬(e1.hasNext() || e2.hasNext())
16:  }
17: ...
18: }

```

(d)

Figure 1: Race-condition in motivating example.

trace to verify the occurrence of a potential concurrency error without manual intervention beyond providing a partial error trace. The technique verifies the race-condition shown in Figure 1 that is reported by Jlint. To our knowledge, this particular race-condition in JDK 1.4 has not been previously reported.

3. GUIDED TEST

We present a technique that guides the test through a sequence of program locations relevant for checking the existence of a particular concurrency error. We guide the execution of the program in a greedy depth-first manner using a two-tier ranking scheme. The states are first ranked based on the number of locations from the given sequence that are encountered along the path. In the case when no new sequence locations are observed the states are ranked using a secondary heuristic value to guide the test toward the next location in the sequence. In this section we describe the input to the guided test, the guided test technique, and the guidance strategy.

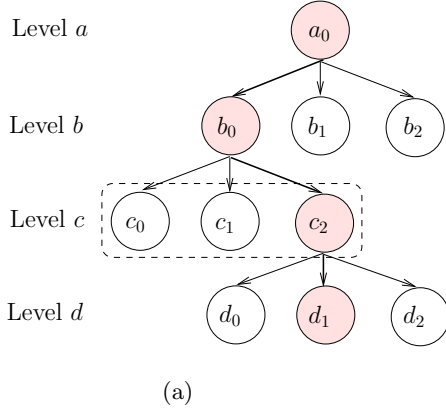
3.1 Input Sequence

The input to our technique is the program, an environment that closes the system, and a sequence of locations that are relevant to the concurrency error in test. The number and type of locations in the sequence can vary based on the concurrency error being tested. For example, to test the occurrences of race-conditions, we can generate a sequence of program locations where unprotected reads and writes occur on shared objects. Tools such as FindBugs [19] and Jlint [1] report suspicious locking patterns that indicate potential race-conditions. Similarly to test deadlocks, based on cyclic lock dependencies, static analysis techniques, such as classical lockset analysis [12, 32], can be used to generate lock-acquisition locations in the program that are part of a cyclic dependency. A larger set of concurrency error patterns are described by Farchi *et. al* in [13]. Understanding and recognizing the concurrent error patterns can be helpful in generating location sequences to test particular errors.

To test the possible race-condition for the example in Figure 1, we need a thread iterating over the list, l_2 , in the `equals` function of `AbstractList` while another thread calls the `add` function. A potential sequence of locations to test this error is as follows:

1. Get the `ListIterator`, e_2 at line 9 in Figure 1(d).
2. Check e_2 `hasNext()` at line 10 in Figure 1(d).
3. Add `some_object` to l_2 at line 10 in Figure 1(b).
4. Call e_2 .`next()` at line 12 in Figure 1(d).

Other pertinent locations can also be added to the sequence. For example, if there is a statement that affects the predicate, `some_condition`, on line 9 in Figure 1(b), we can add the instruction to the sequence. In general, providing as much information as possible in the sequence enables the guided test to be more effective in defect detection.



```

1: /* backtrack := ∅, visited := ∅ */
procedure guided_test( $\langle s, locs, h_{val} \rangle$ )
2:   visited := visited ∪ {s}
3:   while s ≠ null do
4:     if error(s) then
5:       report error statistics
6:     exit
7:      $\langle s, locs, h_{val} \rangle :=$  choose_best_successor( $\langle s, locs, h_{val} \rangle$ )
8:     if s == null then
9:        $\langle s, locs, h_{val} \rangle :=$  get_backtrack_state()

```

Figure 2: Overview of the guided test technique. (a) A graph demonstrating the behavior of the guided test. (b) Pseudocode for the high-level guided test technique.

3.2 Overview of Guided Test

The guided test technique directs the search of the program in a greedy depth-first manner to check whether it leads to an error. We present an overview of the guided test in Figure 2. It is important to note that we use a **visited** set to track explored states. Our study in [25] shows that tracking visited states dramatically increases error discovery rates when compared to stateless search techniques that do not maintain a history of visited states.

Figure 2(a) demonstrates the guided test technique whereas the pseudocode for the guided test technique is presented in Figure 2(b). The input to **guided_test** is the initial state of the program (s), the sequence of locations ($locs$), and the initial heuristic value (h_{val}). In a loop, we guide the test as long as the current state, s , has successors (lines 3 – 9). At every state we check whether the state, s , satisfies the error condition (line 4). If an error is detected, we report the error statistics and exit the test; otherwise, we continue to guide the test. When evaluating a state, the **choose_best_successor** function considers all possible immediate successors of s and assigns to the current state the best successor (line 7). For example, to choose the best successor of b_0 in Figure 2(a), the **choose_best_successor** function ranks c_0 , c_1 , and c_2 (enclosed in a dashed box). As demonstrated by the shaded state in Figure 2(a), c_2 is ranked as the best successor of b_0 . When the test reaches a state with no immediate successors, the technique requests a backtrack state as shown on lines 8–9 in Figure 2(b). The shaded states in Figure 2(a) represent a execution path in the guided test whereas the unshaded states represent the backtrack states. The details of **choose_best_successor** and **get_backtrack_state** are provided in the next section.

3.3 Guidance Strategy

The guidance strategy uses a two-tier ranking scheme. The states are first ranked using a new meta-heuristic which is based on the number of locations in the input sequence that have been encountered along the current execution path. All ties in the meta-heuristic values are randomly broken. Similarly if there a tie at the secondary heuristic level then we,

```

1: /* mStates := ∅, hStates := ∅ */
procedure choose_best_successor( $\langle s, locs, h_{val} \rangle$ )
2:   min_h_val := ∞
3:   for each  $s' \in$  successors(s) do
4:     if  $s' \notin$  visited then
5:       visited := visited ∪ {s'}
6:       locs' := locs
7:       h'_val = get_h_value(s')
8:       if  $s'.current\_loc() == locs.top()$  then
9:         locs'.pop()
10:        mStates := mStates ∪ { $\langle s', locs', h'_{val} \rangle$ }
11:      else if  $h'_{val} == min\_h\_val$  then
12:        hStates := hStates ∪ { $\langle s', locs', h'_{val} \rangle$ }
13:      else if  $h'_{val} < min\_h\_val$  then
14:        hStates := { $\langle s', locs', h'_{val} \rangle$ }
15:        min_h_val := h'_val
16:        backtrack := backtrack ∪ { $\langle s', locs', h'_{val} \rangle$ }
17:   if mStates ≠ ∅ then
18:      $\langle s, locs, h_{val} \rangle :=$  get_random_element(mStates)
19:   else
20:      $\langle s, locs, h_{val} \rangle :=$  get_random_element(hStates)
21:   backtrack := backtrack \ { $\langle s, locs, h_{val} \rangle$ }
22:   bound_size(backtrack)
23:   return  $\langle s, locs, h_{val} \rangle$ 

```

```

procedure get_backtrack_state()
1: if backtrack ≠ ∅ then
2:   x := pick_backtrack_meta_level()
3:   b_points := get_states(backtrack, x)
4:   b_points := b_points ∩ states_min_h_value(b_points)
5:   return get_random_element(b_points)
6: return  $\langle null, \infty, \infty \rangle$ 

```

Figure 3: (a) A two-tier ranking scheme based on the number of locations and the heuristic value (b) Picking a backtrack point.

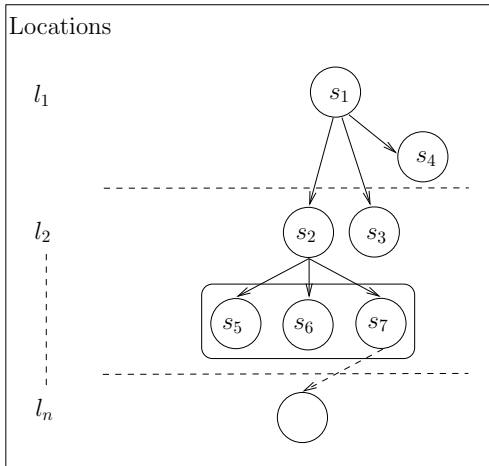


Figure 4: Two-level ranking scheme

again, randomly break the tie. We can use any appropriate existing heuristic to guide the test at the secondary level.

In Figure 3(a) we present the pseudocode to choose the best successor of a given state while guiding the test. The input to the function is a tuple, $\langle s, locs, h_{val} \rangle$ where s is a program state, $locs$ is a sequence of locations, and h_{val} is the heuristic value of s from the secondary heuristic. We evaluate each successor, s' , of s and process s' if it is not found in the `visited` set (line 3 – 4). While processing s' , we add it to the `visited` set (line 5), copy the sequence of locations, $locs$, into a new sequence of locations, $locs'$, and compute the secondary heuristic value for s' (lines 6 – 7). If s' observes an additional location from the sequence then it is added to the `mStates` set (lines 8 – 10); otherwise, it is added to the `hStates` set based whether its h'_{val} is less than the h'_{val} of the other successors of s (lines 11 – 15). The best successor is picked from either the `mStates` or the `hStates` while the other states are added to the `backtrack` set (lines 17 – 22).

We use Figure 4 to demonstrate the two-tier ranking scheme. In Figure 4 the test is guided through locations l_1 to l_n . The dashed-lines separate the states based on the number of locations from the sequence they have observed along the path from the initial state. The states at the topmost level, l_1 , have encountered the first program location in the sequence while states at l_2 have observed the first two program locations from the sequence, and so on and so forth. In Figure 4, state s_1 has three successors: s_2 , s_3 , and s_4 . The states s_2 and s_3 observe an additional location, l_2 , from the sequence compared to their parent, s_1 . When ranking the successors of s_1 , we add the states s_2 and s_3 to the `mStates` set (lines 8 – 10) to denote that a location from the sequence is observed; we also add state s_4 to the `hStates` set (lines 11 – 12). After enumerating the successors of s_1 , the `mStates` set is non-empty (line 17) so we randomly choose between s_2 and s_3 (line 18), and return the state as the best successor. When we evaluate successors of a state that do not encounter any additional location from the sequence, for example, the successors of s_2 in Figure 4 (enclosed by the box), the states are ranked simply based on their secondary heuristic values.

The best successor is then picked from the `hStates` set. All states other than the best successor are added to the `backtrack` set. Note that we bound the size of the `backtrack` set to mitigate the common problem in directed model checking where saving the frontier in a priority queue consumes all memory resources. Bounding the `backtrack` set makes our technique incomplete; however, it is often intractable to obtain complete coverage of large multi-threaded systems.

The `get_backtrack_state` function in Figure 3(b) picks a backtrack point when the guided test reaches the end of a path. The function first checks that the `backtrack` set is not empty. Next, the function probabilistically picks a meta-level, x , between 1 and n where n is the number of locations in the sequence. The states that have observed one program location from the sequence are at meta-level one. We then get all the states at meta level x and return the state with the minimum secondary heuristic value among the states at the same meta-level.

4. STUDY

The study in this paper is designed to evaluate the effectiveness of the guided test approach for detecting concurrency errors in multi-threaded Java programs. The purpose of this study is to evaluate the technique in a test-like paradigm and, thus, we do not compare it with traditional guided search techniques such as best-first and A^* . The study is designed to answer the following three research questions:

- **RQ 1-Defect Detection Rate:** Does the guided test technique increase the rate of defect detection in multi-threaded Java programs when compared to a randomized depth-first search?
- **RQ 2-Effect of the Secondary Heuristic:** What effect does the underlying secondary heuristic have on the effectiveness of the guided test technique in terms of error discovery and use of computation resources?
- **RQ 3-Effect of the Sequence Length:** How does the number of locations provided in the sequence affect the performance of guided test?

4.1 Study Design

We conduct the experiments for our study on a super computing cluster of 618 nodes¹. Every node in the cluster has 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). The execution time for all guided test trials and randomized depth-first search trials, given a particular configuration of a model, is bounded at one hour. We pick this time bound and trials to be consistent with other recent empirical studies [9, 8, 25]. We use the JavaPathfinder (JPF) v4.0 model checker, [31], with partial order reduction turned on as an execution engine for the guided test technique. In the guided test trials we save at most 100,000 backtrack states.

4.2 Artifacts

¹We thank Mary and Ira Lou Fulton for their generous donations to the BYU Supercomputing laboratory

We use six unique multi-threaded Java programs in this study to evaluate the effectiveness of the guided test approach. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM, and classical concurrency errors described in literature [9]. We pick these three artifacts from the benchmark suite because the study in [25] shows that the threads in these programs can be systematically manipulated to create configurations of the model where the error is *hard to find* based on the semantic hardness measure defined in [25]. The particular examples from the benchmark suite were also chosen because they exhibit different concurrency error patterns described by Farchi *et. al* in [13].

The other three examples are programs that use the JDK 1.4 library in accordance with the documentation. Figure 1 is one such program. The particular usage causes various concurrency errors in the JDK 1.4 library. We use Jlint on these models to automatically generate warnings on possible concurrency errors in the JDK 1.4 library. We use the reports to manually generate a short sequence of locations or a partial error trace. We also create parameterized versions of these subjects where certain thread configurations make the error *hard to find* based on the semantic hardness measure of [25]. Note that the guided technique can be applied to the other models in the benchmark suite, [9], and other classes in the JDK 1.4 library which exhibit the same concurrency error patterns as the models chosen in this study. The names, types of models, number of locations used for guidance, and source lines of code (SLOC) are as follows:

TwoStage: Benchmark, Num of locs: 2, SLOC: 52

Reorder: Benchmark, Num of locs: 2, SLOC: 44

Wronglock: Benchmark, Num of locs: 3, SLOC: 38

AbsList: Real, Num of locs: 6, Race-condition in the `AbstractList` class using the synchronized `Vector` sub-class Figure 1. SLOC: 7267

AryList: Real, Num of locs: 6, Race-condition in the `ArrayList` class using the synchronized `List` implementation. SLOC: 7169

Deadlock: Real, Num of locs: 6, Deadlock in the `Vector` and `Hashtable` classes due to a circular data dependency [32]. SLOC: 7151

4.3 Independent Variable

To answer our research questions, we manipulate two independent variables: the underlying secondary heuristics and the number of locations in the sequence. Recall that between meta-levels, where the states do not observe additional locations in the sequence, the guided test uses a secondary heuristic value to guide the search.

We experiment with three different secondary heuristics to study the effect of the underlying heuristic on the effectiveness of the guided test. We experiment with the e-fca heuristic [23], the constant heuristic, and the prefer-thread heuristic [17]. The e-fca heuristic computes the distance between a target program location and the current program location

Subject	Random DFS	Guided Test		
		E-FCA	Const	Prefer Threads
TwoStage(7,1)	0.41	1.00	1.00	1.00
TwoStage(8,1)	0.04	1.00	1.00	1.00
TwoStage(10,1)	0.00	1.00	1.00	1.00
Reorder(9,1)	0.06	1.00	1.00	1.00
Reorder(10,1)	0.00	1.00	1.00	1.00
Wronglock(1,20)	0.28	1.00	1.00	1.00
AbsList(1,7)	0.01	1.00	0.37	0.00
AbsList(1,8)	0.00	1.00	0.08	0.00
Deadlock(1,9)	0.00	1.00	1.00	n/a
Deadlock(1,10)	0.00	1.00	1.00	n/a
AryList(1,5)	0.81	1.00	1.00	1.00
AryList(1,8)	0.00	1.00	1.00	0.01
AryList(1,9)	0.00	1.00	1.00	0.00
AryList(1,10)	0.00	1.00	1.00	0.00

Table 1: Defect Detection Rate.

using the control flow representation of the program. The heuristic based on the distance estimate lends itself naturally to guiding the test toward the next location in the sequence. Note that we modify the heuristic to conservatively estimate the distances in the presence of polymorphism.

The constant heuristic always returns a constant value as the heuristic estimate. It serves as a baseline measure to test the effectiveness of the guided test technique in the absence of any secondary guidance. Recall that since we break all ties randomly, the effect of using the constant heuristic is essentially that of performing a randomized depth-first search with rapid-restarts when the meta-heuristic value does not change.

The prefer-thread heuristic [17] assigns a low heuristic value to a set of user-specified threads. For example, if there are 5 total threads in a program, the user can specify to prefer the execution of certain threads over others when making scheduling choices.

The other independent variable is the sequence length. To test a program we take an initial set of locations to guide the test and then we add or remove locations from the set to measure the effect of the sequence length on the performance of the guided test.

4.4 Dependent Variables and Measure

The dependent variable measured to answer RQ 1 and part of RQ 2 is the error density generated by a technique on a given model. The error density is defined as the probability of a technique finding an error in the program. To compute this probability, we use the ratio of the number of error discovering trials over the total number of trials executed for a given model and technique. A technique that generates an error density of 1.00 is termed effective while a technique that generates an error density of 0.00 is termed ineffective. The other dependent variables we measure for answering the research questions are: the number of states generated, the total time taken before error discovery, the total memory used in error discovery, and the length of the counter-example for the error in the test. Measuring these variables enables us to gauge the effort, in terms of compu-

STATES

Subject	e-FCA Heuristic			Constant Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
TwoStage(7,1)	209	213	217	40851	130839	409156	414187	2206109	4813016
TwoStage(8,1)	246	250	255	49682	217637	502762	609085	4436444	10025314
TwoStage(10,1)	329	333	340	52794	314590	827830	2635251	6690008	8771151
Wronglock(1,10)	804	3526	12542	73	7082	22418	560	120305	675987
Wronglock(1,20)	2445	21391	175708	67	24479	242418	1900	3827020	15112994
Reorder(5,1)	106	109	112	1803	5597	10408	259	977	2402
Reorder(8,1)	193	197	202	17474	36332	65733	523	3110	13536
Reorder(10,1)	266	271	277	28748	67958	110335	771	5136	16492
AryList(1,10)	1764	14044	55241	3652	15972	63206	-	-	-
AbsList(1,10)	1382	1382	1382	10497302	10497302	10497302	-	-	-

TIME IN SECONDS

Subject	e-FCA Heuristic			Constant Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
TwoStage(7,1)	407	449	502	6447	51546	121239	279363	495550	798506
TwoStage(8,1)	415	449	506	11258	86825	166487	491829	760104	1349420
TwoStage(10,1)	424	451	495	14048	128340	259194	498178	912009	1325840
Wronglock(1,10)	693	1544	2198	370	2770	7221	751	19758	89917
Wronglock(1,20)	1307	4422	6247	364	10781	35866	1224	657836	3543484
Reorder(5,1)	290	349	504	1587	2352	3115	434	667	938
Reorder(8,1)	339	447	601	4778	11234	34720	975	1318	2276
Reorder(10,1)	386	464	537	7879	31277	97325	876	1455	2304
AryList(1,10)	1131	3006	5069	2751	7031	24363	-	-	-
AbsList(1,10)	805	921	1133	2584794	2584794	2584794	-	-	-

MEMORY IN MB

Subject	e-FCA Heuristic			Constant Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	163	163	163	261	1050	1863	3715	5324	6259
Twostage(8,1)	163	163	163	437	1501	2223	4536	5421	6450
Twostage(10,1)	163	191	202	631	2011	3328	4915	5636	6356
Wronglock(1,10)	67	104	129	19	109	292	29	292	916
Wronglock(1,20)	89	177	202	19	277	751	49	1538	4282
Reorder(5,1)	163	169	203	50	89	125	19	29	41
Reorder(8,1)	161	167	202	203	419	834	42	61	124
Reorder(10,1)	163	165	203	342	910	1856	39	75	127
AryList(1,10)	247	267	301	154	261	571	-	-	-
AbsList(1,10)	164	178	202	6154	6154	6153	-	-	-

DEPTH OF ERRORS

Subject	e-FCA Heuristic			Constant Heuristic			Prefer-thread Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	38	38	38	52	58	65	52	55	59
Twostage(8,1)	41	41	41	58	63	69	57	60	62
Twostage(10,1)	47	47	47	70	74	79	71	71	71
Wronglock(1,10)	20	21	22	19	28	38	43	49	59
Wronglock(1,20)	20	21	22	23	30	42	22	69	82
Reorder(5,1)	25	25	26	31	36	39	29	30	32
Reorder(8,1)	34	34	34	47	51	55	37	39	41
Reorder(10,1)	40	40	40	54	61	67	41	44	47
AryList(1,10)	85	96	109	99	114	135	-	-	-
AbsList(1,10)	133	133	133	402	402	402	-	-	-

Table 2: Comparison of secondary heuristics when used with the guided test technique.

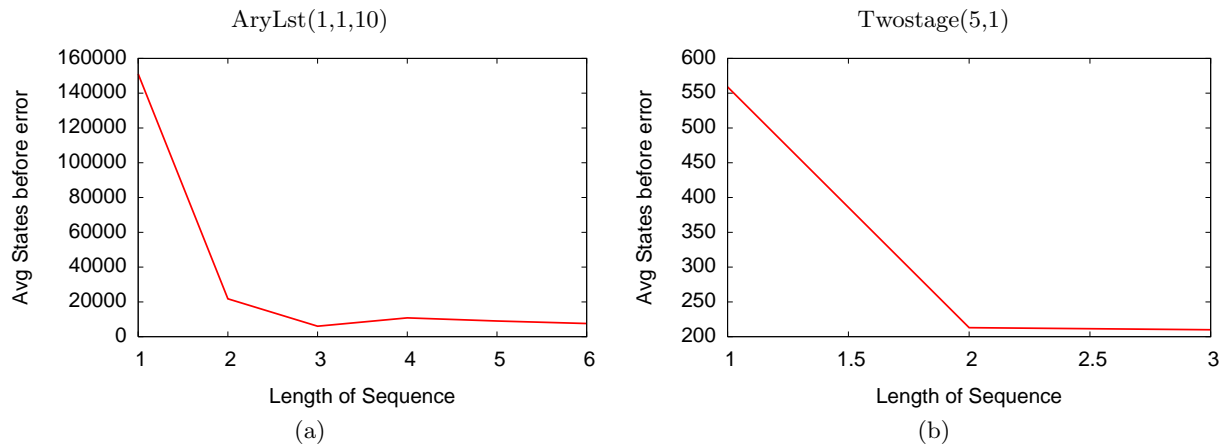


Figure 5: Effect of varying the number of locations in the sequence during guided test using the e-fca as the underlying secondary heuristic. (a) The AryLst(1,1,10) program (b) TwoStage(5,1) benchmark

tation resources, required for error discovery.

5. RESULTS

We present the results of the study and try to answer the three research questions in Table 1, Table 2, and Figure 5.

5.1 RQ 1-Defect Detection Rate

In Table 1, we compare the error densities of randomized depth-first search with the guided test technique in this paper using different underlying secondary heuristics. The first column (**Subject**) in Table 1 specifies the name of the subject and the thread configuration used. The *thread count* in the models is the sum of the numbers specified in the thread configuration. The second column labeled **Random DFS** shows the error densities generated by randomized depth-first search. The last three columns under **Guided Test** are the error densities generated by the guided test technique using a fixed sequence length and three different secondary heuristics: e-fca heuristic (**E-FCA**), constant heuristic (**Const**), and prefer-thread heuristic (**Prefer Threads**). The *n/a* entry denotes that we did not run the test for the particular technique.

The results in Table 1 indicate that guided test, overall, has a better defect detection rate compared to randomized depth-first search. In the different models, as we manipulate the number of threads to increase the semantic hardness of discovering an error in the model (as defined in [25]), the randomized DFS cannot find an error within 100 computation hours. For example, consider the **TwoStage** example where the error density drops from 0.41 to 0.00 when going from the **TwoStage(7,1)** configuration to the **TwoStage(10,1)** configuration. A similar pattern is observed in the **Reorder** model where the error density goes from 0.06 to 0.0; in the **AryList** model the error density drops from a respectable 0.81 to 0.00. For all these models, the guided test technique using the e-fca heuristic finds an error in every single trial as indicated by the error density of 1.00. Even the constant heuristic outperforms the randomized DFS in most models. In the **AbsList** model, however, the constant heuristic has comparatively low error densities of 0.08 and

0.37 in the **AbsList(1,8)** and **AbsList(1,7)** configurations respectively. Among the different underlying heuristics, the prefer-threads heuristic struggles to find an error in the most number of models. The results in Table 1 indicate that guided test, in general, has a better defect detection rate when compared to randomized depth-first search.

5.2 RQ 2-Effect of the Secondary Heuristic

The results in Table 1 show that the choice of the underlying heuristic is important and can affect the performance of the guided test. As the e-fca lends itself naturally to guiding the test toward a location in the program, we notice that the guided test using the e-fca heuristic estimate has the best error discovery rates when compared to guided test trials using the constant and prefer-thread heuristics. The sub-par performance of the prefer-thread heuristic in some models indicates that it is important to choose the secondary heuristic judiciously.

To further evaluate the performance of the different underlying secondary heuristics, we measure four different metrics generated in the error discovering trials of guided test. We measure the number of states explored before error discovery, total time taken for error discovery, the total memory used, and the length of the counter-example. For each metric, we measure the minimum, average and maximum values generated during the 100 trials of guided test. The results are presented in Table 2. Note that the sequence length is fixed in this test.

The first table (labeled **STATES**) in Table 2 reports the minimum, average, and maximum number of states generated in the error discovering trials of guided test using the three secondary heuristics. In the **TwoStage**, **Reorder**, **AryList**, **AbsList** subjects, the minimum, average, and maximum states generated by the e-fca heuristic is perceptibly less than the constant and prefer-thread heuristics. Consider the model, **Twostage(7,1)** where, on average, the e-fca heuristic only generates 213 states while the constant heuristic and prefer-thread heuristic generate 130, 839 and 2, 206, 109 states respectively, on average, before error discovery. In the **AbsList(1,10)** model the e-fca heuristic finds the error every

time by exploring a mere 1382 states. In contrast, only a single guided test, out of 100 trials, using the constant heuristic finds the error after exploring over a million states, while the prefer-thread heuristic is unable to find the error in the 100 trials. `Wronglock` is the only model where the minimum number of states generated by the constant heuristic is less than the e-fca heuristic. This example shows that in certain models, it is possible for the random heuristic to get just lucky. In the `Reorder` model, the prefer-thread heuristic does better than the constant heuristic; however, in all the other models the constant heuristic outperforms the prefer-thread heuristic.

The tables labeled `TIME IN SECONDS` and `MEMORY IN MB` in Table 2 show the minimum, average, and maximum time and memory respectively in the error discovering trials of guided test. The performance patterns described in the previous paragraph for the number of states generated are similar for the total time taken and memory used between the different secondary heuristics. Note that the static analysis performed by the e-fca heuristic to compute distance estimates incurs an additional cost in terms of time and memory compared to the other heuristics. The effect of this additional cost, however, is mitigated because it more effectively guides the test between any two locations in the sequence which leads to a large saving in the total time and memory used. Consider, for example, the `Reorder(5,1)` model where the prefer-thread heuristic takes twice the amount of total time to find an error, on average, while the constant heuristic takes four times as much time, on average, when compared to the e-fca heuristic. The total static analysis time taken by the e-fca heuristic, while conservatively estimating distances in the presence of polymorphism, ranges between two to seven seconds for the results presented in this study.

The table labeled `DEPTH OF ERRORS` in Table 2 shows the minimum, average, and maximum length of the counter-example generated in the error discovering guided test trials. The e-fca is an admissible heuristic; when used in a A^* search, it is guaranteed to find the shortest counter-example. This notion, however, does not hold in the current greedy depth-first search. Interestingly, the length of counter-examples shown in Table 2(d) shows that the e-fca heuristic consistently finds shorter counter-examples when compared to the constant and prefer-thread heuristics.

The results in Table 1 and Table 2, show that guided test using the e-fca heuristic is more effective when compared to the constant and prefer-thread heuristics.

5.3 RQ 3-Effect of the Sequence Length

To answer RQ 3, we vary the number of locations in the sequence for the guided test using the e-fca heuristic as the secondary heuristic. In Figure 5 we plot the average number of states generated before error discovery while varying sequence lengths in two different models. In Figure 5(a) for the `AryList(1,10)` model there is a sharp drop in the number of states when we increase the number of locations from one to two. A smaller decrease in the average number of states is observed between sequence lengths two and three. After three locations, we observe the effects of diminishing returns and the number of states does not vary

much. The same pattern is observed in the `TwoStage` model shown in Figure 5(b). This pattern is observed in most of our models using different secondary heuristics. In general, for the models presented in this study only 2-3 locations are required for guided test to be effective. This is encouraging because it is easy for users to specify 2-3 locations in the program that could possibly lead to a concurrency error. The `AbsList` model with certain thread configurations, however, requires five locations for the guided test to be successful. The `AbsList` model represents the race-condition in the `AbstractList` class while using the synchronized `Vector` sub-class in the JDK 1.4 library from our Figure 1 example. For these, `AbsList` models, the randomized depth-first search has an error density of 0.00. To our knowledge, this is the first report of the error in the JDK 1.4 library.

6. THREATS TO VALIDITY

The subjects used in the study are representative of the other multi-threaded programs in the benchmark suite [9, 25] and JDK 1.4 library. The technique presented in this paper can be applied to other programs of the benchmark suite and other classes in the JDK 1.4 library. We cannot, however, conclusively state whether these artifacts and examples encompass the concurrency errors in a larger set of multi-threaded Java programs. Our study was performed on the JPF model checker. We have already seen that different versions of the same tool yield different results in terms of states generated before error discovery. Specifically we see by comparing results of [9] and [25] that a model that had a *hard to find* error in a previous version of the tool had easily discoverable errors in the new version of the tool which maybe caused by the differences in the implementations of partial order reduction and state storage techniques. This effect is partly mitigated by the fact that if technique A outperforms technique B in one version of the tool, the relation continues to hold in the other version of the tool, regardless of any variance in the measurables.

The guided test requires the secondary heuristic to essentially guide the search toward the next location in the sequence to be effective. The prefer-thread heuristic in its current state, however, does not support the notion of guiding toward a program location which might be the reason for the sub-par performance of the prefer-thread heuristic in the guided test trials. Finally the results presented are also dependent on the variables selected in the study, for example, the number of trials and the execution bound.

7. RELATED WORK

Random Testing [26, 8, 5] uses random inputs or thread schedules to find errors in the program. JCrasher, [5], generates random JUnit test cases for all public methods in a given set of classes. Dwyer *et al.*, [8], randomly sample from different thread-schedules to disperse the search through the state-space and quickly find the error. Randomized search strategies are easily parallelizable and different compute nodes can independently search for the error. Sen, [26], improves the standard randomized search technique by sampling more evenly from the different partial orders. In the study presented in this paper, we specifically choose subjects where randomized test techniques fail to find the error.

Symbolic execution has extensively been used for testing programs and automatic test case generation since the 1970s, [20]. Recent work by Tomb *et al.*, [30], uses symbolic execution to generate concrete paths to null pointer exceptions at an inter-procedural level in sequential programs. Concolic testing, [28], executes the program with random concrete values in conjunction with symbolic execution to collect the path constraints over input data values. jCUTE, [27], in addition to concolic testing, generates thread schedules that execute unique paths in a concurrent program. jCUTE is a white-box testing mechanism that tries to achieve full path-coverage by generating path constraints that make different execution paths feasible. The jCUTE results demonstrate that the cost of constraint solving to achieve full path coverage in a concurrent system is extremely high.

Static analysis techniques ignore the actual execution environment of the program and reason about errors by simply analyzing the source code of the program. Warlock, [29], and ESC/Java, [15], are two static analysis tools that rely heavily on program annotations to find deadlocks and race-conditions in the programs. Annotating existing code is cumbersome and time consuming. RacerX, [12], does a top-down inter-procedural analysis starting from the root of the program. Similarly, the work by Williams *et al.*, [32], does a static deadlock detection for Java libraries. FindBugs [19] and JLint [1], looks for suspicious patterns in Java programs. Error warnings reported by static analysis tools have to be manually verified which is difficult and sometimes not possible. The output of such techniques, however, serve as ideal input for the guided test technique presented in this paper.

Check ‘n’ Crash, [6], is a hybrid test technique that uses a constraint solver to generate concrete test cases based on the output from the static analyzer tool—ESC/Java. Check ‘n’ Crash, however, only generates test cases for safety violations in sequential programs. DSD crasher [7], extends Check ‘n’ Crash by adding information from a runtime analysis tool to ESC/Java to improve its analysis. It too is, however, limited to generating test cases for sequential programs.

Model checking is a formal approach for systematically exploring the behavior of a concurrent software system to verify whether the system satisfies the user specified properties [18, 21, 31, 3]. Counter-example guided abstraction refinement [2] and partial order reduction [14], have been used to partially overcome the explosion in the state-space of the system due to data values and thread-schedules respectively. For concurrent programs, recent seminal work by Cook *et al.*, [4], uses a thread-modular approach to prove termination of concurrent programs. The work, however, does not differentiate between legal termination of a program versus termination due to a deadlock. Another recent work by Gotsman *et al.*, [16], checks for lock consistencies in the presence of deep heap updates using abstract interpretation to detect race-conditions. In contrast to exhaustively searching the system, certain model checking techniques also use heuristics to guide the search quickly toward the error [10, 17, 11, 22, 23]. The state space explosion problem has prevented model checking from scaling to large general concurrent systems. We try to overcome this problem by using the information about a potential error to quickly find the

error.

8. CONCLUSIONS AND FUTURE WORK

This paper presents a guided test technique that automatically verifies potential concurrency errors in multi-threaded Java programs. We provide the guided test a sequence of locations or a partial error trace and the program. The guided test technique automatically controls scheduling decisions to direct the execution of the program using a two-tier ranking scheme. The states are first ranked based on the number of locations, from the given sequence, encountered along the path and then ranked using a secondary heuristic value. The aim of the secondary heuristic value is to guide the test toward the next location in the sequence. The study presented in this paper shows that the guided test technique is effective in error discovery in subjects where randomized depth-first search fails to find an error. Using the guided technique we discovered real concurrency errors in the JDK 1.4 library.

In future work we want to take the output of a static analysis tool and automatically generate a partial error trace and extend the technique to handle non-determinism arising due to data values. The other area open for research is designing a better underlying heuristic. The results in the paper indicate that a better estimate of the distance to the next location in the sequence, overall, improves the performance of a guided test. One problem in current distance heuristics is that they cannot accurately estimate the distances between program locations in the presence of polymorphism.

9. REFERENCES

- [1] C. Artho and A. Biere. Applying static analysis to large-scale, multi-threaded java programs. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, page 68, Washington, DC, USA, 2001. IEEE Computer Society.
- [2] T. Ball, A. Podelski, and S. K. Rajamani. Boolean and cartesian abstraction for model checking C programs. In *TACAS 2001: Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 268–283, London, UK, 2001. Springer-Verlag.
- [3] T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *13th Annual Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
- [4] B. Cook, A. Podelski, and A. Rybalchenko. Proving thread termination. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 320–330, New York, NY, USA, 2007. ACM Press.
- [5] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software - Practice and Experience*, 34(11):1025–1050, 2004.
- [6] C. Csallner and Y. Smaragdakis. Check ‘n’ Crash: combining static checking and testing. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 422–431, New York, NY, USA, 2005. ACM Press.
- [7] C. Csallner and Y. Smaragdakis. DSD-Crasher: a

- hybrid analysis tool for bug finding. In *ISSTA '06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 245–254, New York, NY, USA, 2006. ACM.
- [8] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, pages 3–12, Washington, DC, USA, 2007. IEEE Computer Society.
- [9] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *SIGSOFT '06/FSE-14: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 92–104, New York, NY, USA, 2006. ACM Press.
- [10] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 7th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001.
- [11] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.
- [12] D. Engler and K. Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 237–252, New York, NY, USA, 2003. ACM Press.
- [13] E. Farchi, Y. Nir, and S. Ur. Concurrent bug patterns and how to test them. In *IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 286.2, Washington, DC, USA, 2003. IEEE Computer Society.
- [14] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 110–121, New York, NY, USA, 2005. ACM.
- [15] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.
- [16] A. Gotsman, J. Berdine, B. Cook, and M. Sagiv. Thread-modular shape analysis. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 266–277, New York, NY, USA, 2007. ACM Press.
- [17] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *2002 ACM SIGSOFT International symposium on software testing and analysis*, pages 12–21, 2002.
- [18] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.
- [19] D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004.
- [20] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.
- [21] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.
- [22] N. Rungta and E. G. Mercer. A context-sensitive structural heuristic for guided search model checking. In *20th IEEE/ACM International Conference on Automated Software Engineering*, pages 410–413, Long Beach, California, USA, November 2005.
- [23] N. Rungta and E. G. Mercer. An improved distance heuristic function for directed software model checking. In *FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design*, pages 60–67, Washington, DC, USA, 2006. IEEE Computer Society.
- [24] N. Rungta and E. G. Mercer. Generating counter-examples through randomized guided search. In *Proceedings of the 14th International SPIN Workshop on Model Checking of Software*, pages 39–57, Berlin, Germany, July 2007. Springer-Verlag.
- [25] N. Rungta and E. G. Mercer. Hardness for explicit state software model checking benchmarks. In *5th IEEE International Conference on Software Engineering and Formal Methods*, pages 247–256, London, U.K, September 2007.
- [26] K. Sen. Effective random testing of concurrent programs. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 323–332, New York, NY, USA, 2007. ACM.
- [27] K. Sen and G. Agha. A race-detection and flipping algorithm for automated testing of multi-threaded programs. In *Haifa Verification Conference*, volume 4383 of *Lecture Notes in Computer Science*, pages 166–182. Springer, 2007.
- [28] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 263–272, New York, NY, USA, 2005. ACM.
- [29] N. Sterling. Warlock— a static data race analysis tool. In *USENIX Technical Conference Proceedings*, pages 97–106, 1993.
- [30] A. Tomb, G. Brat, and W. Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA '07: Proceedings of the 2007 international symposium on Software testing and analysis*, pages 97–107, New York, NY, USA, 2007. ACM Press.
- [31] W. Visser, K. Havelund, G. Brat, and S. Park. Model checking programs. In *15th International Conference on Automated Software Engineering (ASE 2000)*, Grenoble, France, September 2000.
- [32] A. Williams, W. Thies, and M. D. Ernst. Static deadlock detection for Java libraries. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 602–629, Glasgow, Scotland, July 27–29, 2005.