A Meta Heuristic for Effectively Detecting Concurrency Errors

Neha Rungta and Eric G Mercer

Department of Computer Science
Brigham Young University
Provo, UT 84602, USA

Abstract. Mainstream programming is migrating to concurrent architectures to improve performance and facilitate more complex computation. The state of the art static analysis tools for detecting concurrency errors are imprecise, generate a large number of false error warnings, and require manual verification of each warning. In this paper we present a meta heuristic to help reduce the manual effort required in the verification of warnings generated by static analysis tools. We manually generate a small sequence of program locations that represent points of interest in checking the feasibility of a particular static analysis warning; then we use a meta heuristic to automatically control scheduling decisions in a model checker to guide the program along the input sequence to test the feasibility of the warning. The meta heuristic guides a greedy depth-first search based on a two-tier ranking system where the first tier considers the number of program locations already observed from the input sequence, and the second tier considers the perceived closeness to the next location in the input sequence. The error traces generated by this technique are real and require no further manual verification. We show the effectiveness of our approach by detecting feasible concurrency errors in benchmarked concurrent programs and the JDK 1.4 concurrent libraries based on warnings generated by 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. 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.

Static analysis tools that analyze the source of the program for detecting concurrency errors are imprecise and incomplete [1–4]. Static analysis techniques are not always useful as they report warnings about errors that *may* exist in the program. The programmer has to manually verify the feasibility of the warning by reasoning about input values, thread schedules, and branch conditions required

to manifest the error along a real execution path in the program. Such manual verification is not tractable in mainstream software development because of the complexity and the cost associated with such an activity.

Model checking in contrast to static analysis is a precise, sound, and complete analysis technique that reports only feasible errors [5, 6]. It accomplishes this by exhaustively enumerating all possible behaviors (state space) of the program to check for the presence and absence of errors; however, the growing complexity of concurrent systems leads to an exponential growth in the size of state space. This state space explosion has prevented the use of model checking in mainstream test frameworks.

Directed model checking focuses its efforts in searching parts of the state space where an error is more likely to exist in order to partially mitigate the state space explosion problem [7–11]. Directed model checking uses heuristic values and path-cost to rank the states in order of interest in a priority queue. Directed model checking uses some information about the program or the property being verified to generate heuristic values. The information is either specified by the user or computed automatically. In this work we use the imprecise static analysis warnings to detect possible defects in the program and use a precise directed search with a meta heuristic to localize real errors.

The meta heuristic presented in this paper guides the program execution in a greedy depth-first manner along an input sequence of program locations. The input sequence is a small number of locations manually generated such that they are relevant in testing the feasibility of a static analysis warning or a reachability property. The meta heuristic ranks the states based on the portion of the input sequence already observed. States that have observed a greater number of locations from the input sequence are ranked as more *interesting* compared to other states. In the case where multiple states have observed the same number of locations in the sequence, the meta heuristic uses a secondary heuristic to guide the search toward the next location in the sequence. In essence, the meta heuristic automatically controls scheduling decisions to drive the program execution along the input sequence in a greedy depth-first manner. The greedy depth-first search picks the best-ranked immediate successor of the current state and does not consider unexplored successors until it reaches the end of a path and needs to backtrack.

In this work we do not consider any non-determinism arising due to data input and only consider the non-determinism arising from thread schedules. 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. The technique is sound in error detection but not complete.

To test the validity of our meta heuristic solution in aiding the process of automatically verifying deadlocks, race conditions, and reachability properties in multi-threaded programs, we present an empirical study conducted on several benchmarked concurrent Java programs and the JDK 1.4 concurrent libraries. We use the Java PathFinder model checker (an explicit state Java byte-code model checker) to conduct the empirical study [6]. We show that the meta

heuristic is extremely effective in localizing a feasible error when given a few key locations relevant to a corresponding static analysis warning. Furthermore, the results demonstrate that the choice of the secondary heuristic has a dramatic effect on the number of states generated, on average, before error discovery.

2 Meta heuristic

In this section we describe the input sequence to the meta heuristic, our greedy depth-first search, and the guidance strategy based on the meta heuristic.

2.1 Input Sequence

The input to our meta heuristic is the program, an environment that closes the program, and a sequence of locations that are relevant to checking the feasibility of the static analysis warning. The number and type of locations in the sequence can vary based on the static analysis warning being verified. For example, to test the occurrences of race-conditions, we can generate a sequence of program locations that represent a series of reads and writes on shared objects. Note that we do not manually specify which thread is required to be at a given location in the input sequence and rely on the meta heuristic to intelligently pick thread assignments.

We use the example in Fig. 1 to demonstrate how we generate an input sequence to check the feasibility of a possible race condition from a static analysis warning. Fig. 1 represents a portion of a program that uses the JDK 1.4 concurrent public library. The `raceCondition` class in Fig. 1(a) initializes two `AbstractList` data structures, l_1 and l_2, using the synchronized `Vector` subclass implementation. 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 Fig. 1(b) on the two threads. The threads go through a sequence of events, including operations on l_1 and l_2 in Fig. 1(b). Specifically, an `add` operation is performed on list l_2 when a certain condition is satisfied; the add is then followed by an operation that checks whether l_1 `equals` l_2. The `add` operation in the `Vector` class, Fig. 1(c), first acquires a lock on its own `Vector` instance and then adds the input element to the instance. 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 Fig. 1(d).

The Jlint static analysis tool issues a series of warnings about potential concurrency errors in the concurrent JDK library when we analyze the program shown in Fig. 1 [4]. The Jlint warnings for the `equals` function in the `AbstractList` class in Fig. 1(d) are on the Iterator operations (lines $8-14$ and lines $18-19$). The warnings state that the Iterator operations are not synchronized. As the program uses a synchronized `Vector` sub-class of the `AbstractList` (in accordance with the specified usage documentation), the user may be tempted

(a)

```
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: }
```

(b)

```
1: class AThread extends Thread{
2:   AbstractList l1;
3:   AbstractList l2;
4:   AThread(AbstractList l1,
5:                 AbstractList l2){
6:     this.l1 := l1; this.l2 := l2;
7:   }
8: public void run(){
9:     ...
10:     if some_condition then
11:        l2.add(some_object);
12:     ...
13:     l1.equals(l2);
14:     ...
15:   }
16: }
```

(c)

```
1: class Vector extends
2:                      AbstractList{
3: ...
4:   public synchronized boolean equals
5:                            (Object o){
6:     super.equals(o);
7:   }
8: ...
9:   public synchronized boolean add
10:                           (Object o){
11:     modCnt ++;
12:     ensureCapacityHelper(cnt + 1);
13:     elementData[cnt ++] = o;
14:     return true;
15:   }
16: ...
17: }
```

(d)

```
1:  class AbstractList
2:                   implements List{
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 :=
10:                  (List o).listIterator();
11:    while e1.hasNext() and
12:                   e2.hasNext() do
13:       Object o1 := e1.next();
14:       Object o2 := e2.next();
15:       if¬(o1 == null ? o2 == null :
16:                   o1.equals(o2)) then
17:          return false;
18:    return ¬(e1.hasNext() ||
19:                   e2.hasNext())
20:  }
21: }
```

Fig. 1. Possible race-condition in the JDK 1.4 concurrent library.

```
1: /* backtrack := ∅, visited := ∅ */
procedure gdf_search(⟨s, locs, h_val⟩)
2: visited := visited ∪ {s}
3: while s ≠ null do
4:    if error(s) then
5:       report error statistics
6:       exit
7:    ⟨s, locs, h_val⟩ := choose_best_successor(⟨s, locs, h_val⟩)
8:    if s == null then
9:       ⟨s, locs, h_val⟩ := get_backtrack_state()
```

Fig. 2. Pseudocode for the greedy depth-first search.

to believe that the warnings are spurious. Furthermore, people most often ignore the warnings in libraries since they assume the libraries to be error-free.

To check the feasibility of the possible race condition reported by Jlint for the example in Fig. 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 input sequence of locations to test the feasibility of the warning is as follows:

1. Get the ListIterator, e_2 at lines $9 - 10$ in Fig. 1(d).
2. Check e_2 `hasNext`() at line 12 in Fig. 1(d).
3. Add `some_object` to l_2 at line 11 in Fig. 1(b).
4. Call e_2.`next`() at line 14 in Fig. 1(d).

The same approach can be applied to generate input sequences for different warnings. Classic lockset analysis techniques detect potential deadlocks in multi-threaded programs caused due to cyclic lock dependencies [2, 12]. For example, it detects a cyclic dependency in the series of lock acquisitions $l_0(A) \rightarrow l_1(B)$ and $l_9(B) \rightarrow l_{18}(A)$, where A and B are the locks acquired at different program locations specified by l_n. To generate an input sequence that checks the feasibility of the possible deadlock we can generate a sequence of locations: $l_0 \rightarrow l_9 \rightarrow l_1 \rightarrow l_{18}$. 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 detect particular errors.

In general, providing as much relevant information as possible in the sequence enables the meta heuristic to be more effective in defect detection; however, only 2–3 key locations were required to find errors in most of the models in our study. Any program location that we think affects the potential error can be added to the sequence. For example, if there is a data definition in the program that affects the variables in the predicate, `some_condition`, of the branch statement shown on line 10 in Fig. 1(b), then we can add the program location of the data definition to the sequence. Similarly we can generate input sequences to check reachability properties such as NULL pointer exceptions and assertion violations in multi-threaded programs.

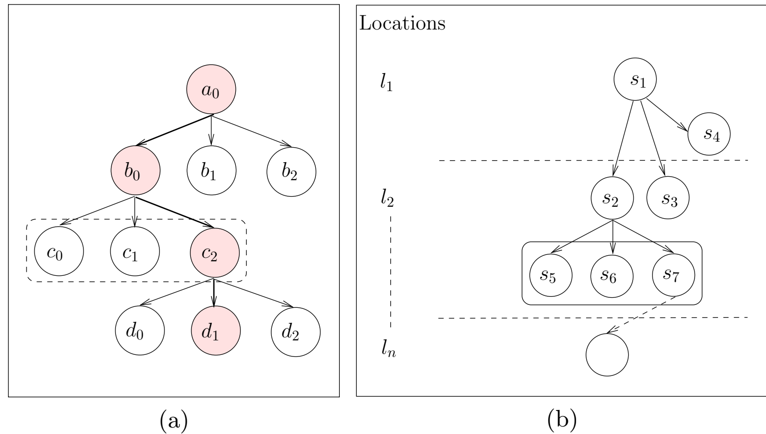

Fig. 3. Guidance (a) Greedy depth-first search (b) Two-level ranking scheme

2.2 Greedy depth-first search

In this subsection we describe a greedy depth-first search that lends itself naturally in directing the search using the meta heuristic along a particular path (the input sequence of locations). The greedy depth-first search mimics a test-like paradigm for multi-threaded programs. The meta heuristic can be also used with bounded priority-queue based best-first searches with comparable results.

The pseudocode for the greedy depth-first search is presented in Fig. 2. The input to `gdf_search` is a tuple with the initial state of the program (s), the sequence of locations ($locs$), and the initial secondary heuristic value (h_{val}). In a loop we guide the execution 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, then we report the error details and exit the search; otherwise, we continue to guide the search. The `choose_best_successor` function only considers the immediate successors of s and assigns to the current state the best-ranked successor of s (line 7). When the search reaches a state with no immediate successors, the technique requests a backtrack state as shown on lines $8 - 9$ in Fig. 2. The details of `choose_best_successor` and `get_backtrack_state` are provided in Fig. 4 and Fig. 5 respectively.

Fig. 3(a) demonstrates the greedy depth-first search using a simple example. The `choose_best_successor` function ranks c_0, c_1, and c_2 (enclosed in a dashed box) to choose the best successor of b_0 in Fig. 3(a). The shaded state c_2 is ranked as the best successor of b_0. When the search reaches state d_2 that does not have any successors, the search backtracks to one of the unshaded states (e.g., b_1, b_2, c_0, c_1, d_0, or d_2). We bound the number of unshaded states (backtrack states) saved during the search. Bounding the backtrack states makes our technique incomplete; although, the bounding is not a limitation because obtaining a complete coverage of the programs we are considering is not possible.

```
 1: /* mStates := ∅, hStates := ∅, min_h_val := ∞ */
procedure choose_best_successor(⟨s, locs, h_val⟩)
 2: for each s' ∈ successors(s) do
 3:   if ¬visited.contains(s') then
 4:     visited.add_state(s')
 5:     locs' := locs /* Make copy of locs */
 6:     h'_val = get_h_value(s')
 7:     if s'.current_loc() == locs.top() then
 8:       mStates := next_state_to_explore(mStates, ⟨s', locs'.pop(), h'_val⟩)
 9:     else
10:       hStates := next_state_to_explore(hStates, ⟨s', locs, h'_val⟩)
11:     backtrack.add_state(⟨s', locs', h'_val⟩)
12: if mStates ≠ ∅ then
13:   ⟨s, locs, h_val⟩ := get_random_element(mStates)
14: else
15:   ⟨s, locs, h_val⟩ := get_random_element(hStates)
16: backtrack.remove_state(⟨s, locs, h_val⟩)
17: bound_size(backtrack)
18: return ⟨s, locs, h_val⟩

procedure next_state_to_explore(states, ⟨s, locs, h_val⟩)
 1: if states == ∅ or h_val == min_h_val then
 2:   states.add_state(⟨s, locs, h_val⟩)
 3: else if h_val < min_h_val then
 4:   states.clear()
 5:   states.add_state(⟨s, locs, h_val⟩)
 6:   min_h_val := h_val
 7: return states
```

Fig. 4. Two-tier ranking scheme for the meta heuristic.

2.3 Guidance Strategy

The meta heuristic uses a two-tier ranking scheme as the guidance strategy. The states are first assigned a rank based on the number of locations in the input sequence that have been encountered along the current execution path. The meta heuristic then uses a secondary heuristic to rank states that observed the same number of locations in the sequence. The secondary heuristic is essentially used to guide the search toward the next location in the input sequence.

In Fig. 4 we present the pseudocode to choose the best successor of a given state. 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 generated by the secondary heuristic function. We evaluate each successor of s, s', and process s' if it is not found in the `visited` set (line $2-3$). To process s' we add it to the `visited` set (line 4), copy the sequence of locations *locs* into a new sequence of locations $locs'$ (line 5), and compute the secondary heuristic value for s' (line 6). If s' observes an additional location from the sequence (line 7), then we update the `mStates` set (line 8); otherwise, we update the `hStates` set (line 10). An element from the $locs'$ is removed on line 8 to indicate s' has observed an additional location. We invoke the `next_state_to_explore` function with the `mStates` or the `hStates` set and the tuple containing s'. The best successor is

```
procedure get_backtrack_state()
1: if backtrack == ∅ then
2:     return ⟨null, ∞, ∞⟩
3: else
4:     x := pick_backtrack_meta_level()
5:     b_points := get_states(backtrack, x)
6:     b_points := b_points ∩ states_min_h_value(b_points)
7:     return get_random_element(b_points)
```

Fig. 5. Stochastic backtracking technique.

picked from `mStates` if it is non-empty; else, it is picked from the `hStates` set. The algorithm prefers states in the `mStates` set because they have observed an additional location compared to their parent. All other successor states are added to the `backtrack` set (lines $12 - 18$).

The `next_state_to_explore` function in Fig. 4 uses the secondary heuristic values (h_{val}) to add states to the `mStates` and `hStates` sets. Recall that the `next_state_to_explore` is invoked with either the `mStates` set or `hStates` set which is mapped to the formal parameter `states`. When the `states` set is empty or the h_{val} is equal to the minimum heuristic value (min_h_{val}) then the algorithm simply adds the tuple with the successor state to the `states` set. If, however, the h_{val} is less than the minimum heuristic value then the algorithm clears the `states` set, adds the tuple with the successor state to `states`, and sets the value of min_h_{val} to h_{val}. Finally, the function returns the `states` set.

We use Fig. 3(b) to demonstrate the two-tier ranking scheme. In Fig. 3(b) the search 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, so on and so forth. In Fig. 3(b) we see that 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. Suppose s_2 and s_3 have the same secondary heuristic value. We add the states s_2 and s_3 to the `mStates` set to denote that a location from the sequence is observed. Suppose, the secondary heuristic value of s_4 is greater than that of s_2 and s_3; then s_4 is not added to the `hStates` set.

After enumerating the successors of s_1, the `mStates` set is non-empty so we randomly choose between s_2 and s_3 (line 13 in Fig. 4) 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 Fig. 3(b) (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. 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.

The get_backtrack_state function in Fig. 5 picks a backtrack point when the guided test reaches the end of a path. Backtracking allows the meta heuristic to pick a different set of threads when it is unable to find an error along the initial sequence of thread schedules. As shown in Fig. 5, if the backtrack set is empty, then the function returns null as the next state (lines $1-2$); otherwise, 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 that meta level. The stochastic element of picking backtrack points enables the search to avoid getting stuck in a local minima.

3 Empirical Study

The empirical study in this paper is designed to evaluate the effectiveness of the meta heuristic in detecting concurrency errors in multi-threaded Java programs.

3.1 Study Design

We conduct the experiments on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of greedy depth-first search and randomized depth-first search. All the trials are bounded at one hour. We execute multiple trials of the greedy depth-first search since all ties in heuristic values are brokenly randomly and there is a stochastic element in picking backtrack points. An extensive study shows that randomly breaking ties in heuristic values helps in overcoming the limitations (and benefits) of default search order in directed search techniques [14]. We pick the time bound and number of trials to be consistent with other recent empirical studies [15–17]. Since each trial is completely independent of the other trials we use a super computing cluster of 618 nodes to distribute the trials on various nodes and quickly generate the results.[1] We use the Java Pathfinder (JPF) v4.0 Java byte-code model checker with partial order reduction turned on to run the experiments [6]. In the greedy depth-first search trials we save at most 100,000 backtrack states.

We use six unique multi-threaded Java programs in this study to evaluate the effectiveness of the meta heuristic in checking whether the input sequence leads to an error. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM Research Lab in Haifa, and classical concurrency errors described in literature [15]. We pick these three artifacts from the benchmark suite because the threads in these programs can be systematically manipulated to create configurations of the model where randomized depth-first

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

Subject	Total Threads	Random DFS	Meta Heuristic		
			PFSM	Rand	Prefer Threads
TwoStage(7,1)	9	0.41	1.00	1.00	1.00
TwoStage(8,1)	10	0.04	1.00	1.00	1.00
TwoStage(10,1)	12	0.00	1.00	1.00	1.00
Reorder(9,1)	11	0.06	1.00	1.00	1.00
Reorder(10,1)	12	0.00	1.00	1.00	1.00
Wronglock(1,20)	22	0.28	1.00	1.00	1.00
AbsList(1,7)	9	0.01	1.00	0.37	0.00
AbsList(1,8)	10	0.00	1.00	0.08	0.00
Deadlock(1,9)	11	0.00	1.00	1.00	1.00
Deadlock(1,10)	12	0.00	1.00	1.00	1.00
AryList(1,5)	7	0.81	1.00	1.00	1.00
AryList(1,8)	10	0.00	1.00	1.00	0.01
AryList(1,9)	11	0.00	1.00	1.00	0.00
AryList(1,10)	12	0.00	1.00	1.00	0.00

Table 1. Error density of the models with different search techniques.

search is unable to find errors in the models [17]. These models also 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. Fig. 1 is one such program that appears as `AbsList` in our results. We use Jlint on these models to automatically generate warnings on possible concurrency errors in the JDK 1.4 library and then manually generate the input sequences. The name, type of model, number of locations in the input sequence, and source lines of code (SLOC) for the models 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 Fig. 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 [12]. SLOC: 7151

3.2 Error Discovery

In Table 1 we compare the error densities of randomized depth-first search (`Random DFS`) to the meta heuristic using a greedy depth-first search. The error density which is a dependent variable in this study 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

Subject	PFSM Heuristic			Random 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	-	-	-

Table 2. Comparison of the heuristics when used with the meta heuristic.

trials executed for a given model and technique. A technique that generates an error density of 1.00 is termed effective in error discovery while a technique that generates an error density of 0.00 is termed ineffective for error discovery.

We test three different secondary heuristics which is an independent variable to study the effect of the underlying heuristic on the effectiveness of the meta heuristic: (1) The polymorphic distance heuristic (`PFSM`) computes the distance between a target program location and the current program location on the control flow representation of the program. The heuristic rank based on the distance estimate lends itself naturally to guiding the search toward the next location in the sequence [11]. (2) The random heuristic (`Rand`) always returns a random value as the heuristic estimate. It serves as a baseline measure to test the effectiveness of guiding along the input sequence in the absence of any secondary guidance. (3) The prefer-thread heuristic (`Prefer Threads`) assigns a low heuristic value to a set of user-specified threads [8]. For example, if there are five total threads in a program then the user can specify to prefer the execution of certain threads over others when making scheduling choices.

The results in Table 1 indicate that the meta heuristic, overall, has a higher error discovery rate compared to randomized depth-first search. In the `TwoStage` example the error density drops from 0.41 to 0.00 when going from the configuration of `TwoStage(7,1)` 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 meta heuristic using the polymorphic distance heuristic finds an error in every single trial as indicated by the error density of 1.00. In some cases, even when we use the random heuristic as the secondary heuristic, the greedy depth-first search outperforms the randomized depth-first search.

The `AbsList`, `AryList`, and `Deadlock` models represent real errors in the JDK 1.4 concurrent library. The `AbsList` model contains the portion of code shown in Fig. 1. In addition to the locations shown in Section 2.1 we manually add other data definition locations that are relevant in reaching the locations shown in Section 2.1. We use the meta heuristic to successfully generate

a concrete error trace for the possible race condition reported by Jlint. The counter-example shows that the race-condition is caused because the `equals` method in Fig. 1(c) never acquires a lock on the input parameter. This missing lock allows another thread to modify the list (by adding an object on line 11 in Fig. 1(b)) while the thread is iterating over the list in the `equals` method. To our knowledge, this is the first report of the particular race condition in the JDK 1.4 library. It can be argued that the application using the library is incorrect and changing the comparison on line 13 of Fig. 1(b) to l_2.`equals`(l_1) can fix the error; however, we term it as a bug in the library because the usage of the library is in accordance with the documentation.

Table 2 reports the minimum, average, and maximum number of states generated in the error discovering trials of the meta heuristic using the three secondary heuristics. The entries in Table 2 marked "-" indicate that the technique was unable to find an error in 100 independent greedy depth-first search trials that are time-bounded at one hour. In the `TwoStage`, `Reorder`, `AryList`, `AbsList` subjects, the minimum, average, and maximum states generated by the PFSM heuristic is perceptibly less than the random and prefer-thread heuristics. Consider the `Twostage(7,1)` model where, on average, the PFSM heuristic only generates 213 states while the random 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 PFSM heuristic finds the error every time by exploring a mere 1382 states. In contrast, from a total of 100 trials with the random heuristic only a single trial 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 random heuristic is less than the PFSM heuristic. This example shows that it is possible for the random heuristic to get just lucky in certain models. The results in Table 2 demonstrate that a better underlying secondary heuristic helps the meta heuristic generate fewer states before error discovery. The trends observed in Table 2 are also observed in total time taken before error discovery, total memory used, and length of counter-example.

3.3 Effect of the Sequence Length

We vary the number of key locations in the input sequence provided to the meta heuristic to study the effect of the number of locations on the performance of the meta heuristic. In Fig. 6 we plot the average number of states generated (the dependent variable) before error discovery while varying sequence lengths in the `AryLst` model. In Fig. 6 there is a sharp drop in the number of states when we increase the number of key locations from one to two. A smaller decrease in the average number of states is observed between sequence lengths two and three. We observe the effects of diminishing returns after three key locations and the number of states does not vary much. In general, for the models presented in this study, only 2–3 key locations are required for the meta heuristic to be effective. In the possible race condition shown in Fig. 1 (`AbsList` model), however, we needed to specify a minimum of five key program locations in the input sequence for

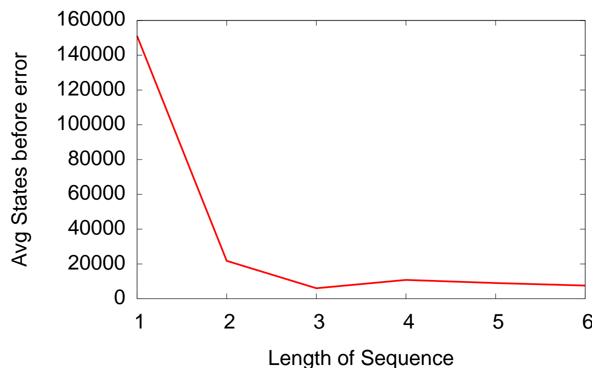

Fig. 6. Effect of varying the number of locations in the sequence in the AryLst(1,10) program to verify the race condition in the JDK1.4 concurrent library.

the meta heuristic to find a corresponding concrete error trace. Recall that the `AbsList` model represents the race-condition in the `AbstractList` class while using the synchronized `Vector` sub-class in the JDK 1.4 library.

4 Related Work

Static analysis techniques ignore the actual execution environment of the program and reason about errors by simply analyzing the source code of the program. ESC/Java relies heavily on program annotations to find deadlocks and race-conditions in the programs [1]. Annotating existing code is cumbersome and time consuming. RacerX does a top-down inter-procedural analysis starting from the root of the program [2]. Similarly, the work by Williams *et al.* does a static deadlock detection in Java libraries [12]. FindBugs and Jlint look for suspicious patterns in Java programs [3, 4]. 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 meta heuristic presented in this paper. Furthermore, dynamic analysis techniques can also be used to generate warnings about potential errors in the programs [18, 19].

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 [5, 6]. In contrast to exhaustively searching the system, directed model checking uses heuristics to guide the search quickly toward the error [7, 8, 20, 9–11]. Property-based heuristics and structural heuristics consider the property being verified and structure of the program respectively to compute a heuristic rank [7, 8]. Distance estimate heuristics rank the states based on the distance to a possible error location [20, 9–11]. As seen in the results, the PFSM distance heuristic is very effective in guiding the search toward a particular lo-

cation; however, its success is dramatically improved in combination with the meta heuristic.

The trail directed model checking by Edelkamp *et. al* uses a concrete counter-example generated by a depth-first search as input to its guidance strategy [21]. It uses information from the original counter-example (trail) in order to generate an optimal counter-example. The goal in this work, however, is to achieve error discovery in models where exhaustive search techniques are unable to find an error. The deterministic execution technique used to test concurrent Java monitors is related to the technique presented in this paper [22]. The deterministic execution approach, however, requires a significant manual effort with the tester required to provide data values to execute different branch conditions, thread schedules, and sequence of methods.

Similar ideas of guiding the program execution using information from some abstraction of the system have been explored in hardware verification with considerable success [23, 24]. An interesting avenue of future work would be to study the reasons for the success (in concretizing abstract traces by guiding program execution) that we observe in such disparate domains with very different abstraction and guidance strategies.

5 Conclusions and Future Work

This paper presents a meta heuristic that automatically verifies the presence of errors in real multi-threaded Java programs based on static analysis warnings. We provide the meta heuristic a sequence of locations and it automatically controls scheduling decisions to direct the execution of the program using a two-tier ranking scheme in a greedy-depth first manner. The study presented in this paper shows that the meta heuristic is effective in error discovery in subjects where randomized depth-first search fails to find an error. Using the meta heuristic 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 the input sequence using control and data dependence analyses. Also we would like to extend the technique to handle non-determinism arising due to data values.

References

1. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: Proc. PLDI, New York, NY, USA, ACM (2002) 234–245
2. Engler, D., Ashcraft, K.: RacerX: effective, static detection of race conditions and deadlocks. In: SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles, New York, NY, USA, ACM Press (2003) 237–252
3. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12) (2004) 92–106
4. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded java programs. In: Proc. ASWEC, Washington, DC, USA, IEEE Computer Society (2001) 68

5. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
6. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: Proc. ASE, Grenoble, France (September 2000)
7. Edelkamp, S., Lafuente, A.L., Leue, S.: Directed explicit model checking with HSF-SPIN. In: Proc. SPIN Workshop. Number 2057 in Lecture Notes in Computer Science, Springer-Verlag (2001)
8. Groce, A., Visser, W.: Model checking Java programs using structural heuristics. In: Proc. ISSTA. (2002) 12–21
9. Rungta, N., Mercer, E.G.: A context-sensitive structural heuristic for guided search model checking. In: Proc. ASE, Long Beach, California, USA (November 2005) 410–413
10. Rungta, N., Mercer, E.G.: An improved distance heuristic function for directed software model checking. In: Proc. FMCAD, Washington, DC, USA, IEEE Computer Society (2006) 60–67
11. Rungta, N., Mercer, E.G.: Guided model checking for programs with polymorphism. In: Proc. PEPM, New York, NY, USA, ACM (2008) 21–30
12. Williams, A., Thies, W., Ernst, M.D.: Static deadlock detection for Java libraries. In: ECOOP 2005 — Object-Oriented Programming, 19th European Conference, Glasgow, Scotland (July 27–29, 2005) 602–629
13. Farchi, E., Nir, Y., Ur, S.: Concurrent bug patterns and how to test them. In: IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing, Washington, DC, USA, IEEE Computer Society (2003) 286.2
14. Rungta, N., Mercer, E.G.: Generating counter-examples through randomized guided search. In: Proceedings of the 14th International SPIN Workshop on Model Checking of Software, Berlin, Germany, Springer–Verlag (July 2007) 39–57
15. Dwyer, M.B., Person, S., Elbaum, S.: Controlling factors in evaluating path-sensitive error detection techniques. In: Proc. FSE '06, New York, NY, USA, ACM Press (2006) 92–104
16. Dwyer, M.B., Elbaum, S., Person, S., Purandare, R.: Parallel randomized state-space search. In: Proc. ICSE '07, Washington, DC, USA, IEEE Computer Society (2007) 3–12
17. Rungta, N., Mercer, E.G.: Hardness for explicit state software model checking benchmarks. In: Proc. SEFM '07, London, U.K (September 2007) 247–256
18. Havelund, K.: Using runtime analysis to guide model checking of java programs. In: Proc. SPIN Workshop, London, UK, Springer-Verlag (2000) 245–264
19. Shacham, O., Sagiv, M., Schuster, A.: Scaling model checking of dataraces using dynamic information. J. Parallel Distrib. Comput. **67**(5) (2007) 536–550
20. Edelkamp, S., Mehler, T.: Byte code distance heuristics and trail direction for model checking Java programs. In: Proc. MoChArt. (2003) 69–76
21. Edelkamp, S., Lafuente, A.L., Leue, S.: Trail-directed model checking. In Stoller, S.D., Visser, W., eds.: ENTCS. Volume 55., Elsevier Science Publishers (2001)
22. Harvey, C., Strooper, P.: Testing Java monitors through deterministic execution. In: Proc. ASWEC, Washington, DC, USA, IEEE Computer Society (2001) 61
23. Nanshi, K., Somenzi, F.: Guiding simulation with increasingly refined abstract traces. In: Proc. DAC, New York, NY, USA, ACM (2006) 737–742
24. Paula, F.M.D., Hu, A.J.: An effective guidance strategy for abstraction-guided simulation. In: Proc. DAC '07, New York, NY, USA, ACM (2007) 63–68