Guided Model Checking for Programs with Polymorphism

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

Eric G. Mercer
Computer Science Department,
Brigham Young University,
Provo, UT 84601, USA
egm@cs.byu.edu

Abstract

Exhaustive model checking search techniques are ineffective for error discovery in large and complex multi-threaded software systems. Distance estimate heuristics guide the concrete execution of the program toward a possible error location. The estimate is a lower-bound computed on a statically generated abstract model of the program that ignores all data values and only considers control flow. In this paper we describe a new distance estimate heuristic that efficiently computes a tighter lower-bound in programs with polymorphism when compared to the state of the art distance heuristic. We statically generate conservative distance estimates and refine the estimates when the targets of dynamic method invocations are resolved. In our empirical analysis the state of the art approach is computationally infeasible for large programs with polymorphism while our new distance heuristic can quickly detect the errors.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking

General Terms Reliability, Verification

Keywords Software model checking, guided search, heuristics, error discovery

1. Introduction

The ubiquity of multi-core processors is creating a paradigm shift from inherently sequential to highly concurrent and parallel systems. The lack of scalable verification techniques to detect concurrency errors is proving to be a hindrance for programmers developing concurrent programs. The trend toward parallelism and concurrency motivates a need to develop effective and scalable error detection techniques for concurrent programs.

Model checking techniques exhaustively enumerate all possible behaviors of the system to verify the presence as well as the absence of errors in programs (Ball and Rajamani 2001; Henzinger et al. 2003; Holzmann 2003; Robby et al. 2003; Visser et al. 2003). The systematic exploration of all possible behaviors enables model checking to find subtle concurrency errors that are often missed by ad-hoc testing techniques. The exhaustive nature of model checking leads to a huge state space explosion making it intractable in verifying practical applications.

Guided model checking tries to overcome the state space explosion problem by focusing the search in parts of the program that are more likely to contain an error (Edelkamp et al. 2001b; Edelkamp and Mehler 2003; Groce and Visser 2002; Rungta and Mercer 2005, 2006). Guided model checking techniques use heuristic functions to rank states in order of interest in an attempt to quickly generate a counterexample. States are ordered in a priority queue or a search stack based on their heuristic rank and path cost such that the states estimated to lead to an error state are explored before others.

Distance estimate heuristics try to compute a reasonable lower-bound on the number of computation steps required to reach a target location from the current location (Edelkamp and Mehler 2003; Rungta and Mercer 2005, 2006). The distance estimates are used to guide the concrete program execution toward the target locations. The target locations are either provided by the user or generated using static analysis techniques. The estimates are computed on a statically generated abstract model that ignores all data values and only considers the control flow of the program. In the absence of data values to compute an accurate distance estimate between two arbitrary program locations is undecidable in general.

The FSM distance heuristic is the state of the art distance heuristic for programs with polymorphism (Edelkamp and Mehler 2003). The FSM distance heuristic ignores all calling context information and is unable to compute a reasonable lower-bound. Furthermore, the complexity of the FSM distance heuristic is cubic in the total number of instructions in the program. This complexity renders the FSM distance heuristic intractable for computing distance estimates in medium to large sized programs. Note that this is after we perform a rapid type analysis that uses the information about instantiated classes to create a reduced set of executable methods in programs with polymorphism (Bacon and Sweeney 1996).

In this work we present a distance heuristic estimate that computes a tighter lower-bound on the distance estimates in polymorphic programs compared to the FSM distance heuristic. The polymorphic distance heuristic first performs an interprocedural static analysis to compute an initial lower-bound on distance estimates; second, during the model checking we refine the distance estimates on demand when the targets of dynamic method invocations are resolved. The complexity of the new approach is cubic in the number of instructions in the method with the largest number of instructions. Note that this is significantly less than the complexity of the FSM distance heuristic.

We present an empirical analysis to demonstrate the effectiveness of the polymorphic distance heuristic in error discovery in a set of benchmarked multi-threaded programs where exhaustive and randomized search techniques are unable to find an error. We compare the polymorphic distance heuristic to the FSM distance heuristic and a random heuristic that assigns a random value as the rank of a state. Using the polymorphic distance heuristic we are able to detect real errors in the JDK 1.4 concurrent library. We also demonstrate that it significantly outperforms the FSM distance heuristic and the random heuristic in the number of states, time taken, and total memory used before error discovery.

2. Background

Distance estimate heuristics compute a heuristic value based on the distance to a target state, t, from a current state, s. The state s contains a set of unique thread identifiers, a program location and stack for each thread, and a heap. A transition relation generates a set of successor states for s, $\{s'_0, s'_1, \ldots, s'_n\}$, where the transition to each successor $s \rightarrow s'_i$ represents a possible change in state from s. Iteratively applying the transition relation to each state allows us to build the entire reachable behavior space of the program. A path, π, is a sequence of transitions, $s \rightarrow s' \rightarrow s'' \rightarrow s''' \ldots$, that represents a feasible execution path in the behavior space.

Definition 2.1. *The distance, d, between state, s, and target state, t, is the number of computation steps in an execution path from s to t:* $d(s,t) := |\pi|$ *where* $\pi := s \rightarrow s' \rightarrow s'' \rightarrow \ldots \rightarrow t$.

Computing accurate distance estimates between two states requires us to first build the reachable state space of a program that essentially entails solving the original problem a priori. In order to overcome this problem, distance estimate heuristics use a heuristic function, h, that approximates the distance between s and t on a statically generated abstract transition graph of the program. In the abstract system a state simply contains a unique program location identifier. In other words, an abstract state represents a single program instruction. The control flow of the program is the transition relation used to generate the abstract transition system. The abstract transition system ignores all data values and is an over-approximation of the original system. The program location of the recently executed thread in state s is used to map the concrete state to an abstract state. The heuristic function, h, estimates the distance between s and t by computing the distance between their respective abstract counterparts in the abstract transition system. Different distance heuristics compute the heuristic values on the abstract transition graph with varying degrees of calling context information.

The FSM distance heuristic is the state of the art heuristic for computing distance estimates in programs with polymorphism (Edelkamp and Mehler 2003). The FSM distance heuristic performs an interprocedural control flow analysis to statically compute the lower-bound on the distance between two arbitrary instructions in the program. The FSM distance heuristic is unable to compute a reasonable lower-bound because it ignores all calling context information and simply minimizes across the different methods. A detailed example is shown in (Rungta and Mercer 2005). This problem is further exacerbated in programs with polymorphism because it minimizes the distance estimates across all implementing sub-type targets of a dynamic method invocation.

The e-FCA heuristic computes full calling context-aware distance estimates in non-recursive C programs with resolved function pointers using a combination of static and dynamic information (Rungta and Mercer 2006). The e-FCA improves on previous distance estimates based on the FSM heuristic function and the EFSM heuristic function (Edelkamp and Mehler 2003; Rungta and Mercer 2005). The FSM distance heuristic does not consider any calling context while the EFSM distance heuristic only considers partial context information. A comparative empirical study in (Rungta and Mercer 2006) demonstrates that computing a tighter lower-bound by adding more calling context information enables us to more efficiently find an error.

The e-FCA distance estimate is computed based on the statically generated abstract model that only contains control flow information with following rules: at a given program location, we can either (a) reach the return statement of the current function and return to its caller without encountering the target location; or (b) reach the target location without executing the return statement of the current function (the target location can be reached in the forward direction). In cases where the target location cannot be reached in the forward direction, the e-FCA looks up the return point of the current function on the dynamic runtime stack of the recently executed thread in the concrete state. If the target location is reachable in the forward direction from the return point then the final estimate is the summation of the cost of moving to the return point and the distance in the forward direction from the return point to the target location. Otherwise, the algorithm keeps unrolling the stack until it reaches the `main` function.

The e-FCA lower-bounds all distance estimates by computing the shortest paths through various branching and looping constructs of a program. This allows the heuristic to be admissible and consistent.

Definition 2.2. *An admissible heuristic h is a function that guarantees a lower bound on the distance from every state, s, to the target state, t:* $h(s,t) \leq d(s,t)$.

Definition 2.3. *A consistent heuristic h is a function that guarantees for every state s and each successor of s′ of s the estimated distance from s to t is less than or equal to the distance between s and s′ plus the estimated distance from s′ to t:* $h(s,t) \leq d(s,s') + h(s',t)$

In an A^* search, (Russell et al. 1995), the e-FCA generates minimal length counter-examples. The e-FCA heuristic is, however, not designed to compute distance estimates in the presence of dynamic method invocations whose targets cannot be statically resolved using a type analysis.

3. Motivation

It is important to compute accurate distance estimates in the presence of polymorphism because there is an increasing use of object oriented languages like Java and C# which inherently encourage the use of polymorphism. More importantly, Java and C# are being used to develop concurrent applications because they natively support concurrency. The inability of the FSM distance heuristic to compute estimates in the presence of polymorphism makes it ineffective for guiding program execution in Java and C# programs.

The example shown in Figure. 1 is the `equals` function of the `AbstractList` class in the JDK 1.4 concurrent library. In order

```
class AbstractList implements List{
...
public boolean equals(Object $o$){
  if $o$ == this then
      return true;
  if ¬($o$ instanceof List) then
      return false;
  ListIterator $e_1$ := ListIterator();
  ListIterator $e_2$ := (List $o$).listIterator();
  while $e_1$.hasNext() and $e_2$.hasNext() do
      Object $o_1$ := $e_1$.next();
      Object $o_2$ := $e_2$.next();
      if¬($o_1$ == null ? $o_2$ == null : $o_1$.equals($o_2$))then
          return false;
  return ¬($e_1$.hasNext() || $e_2$.hasNext())
}
...
}
```

Figure 1. The `equals` function in the `AbstractList` implementation of the JDK 1.4 library which uses polymorphism.

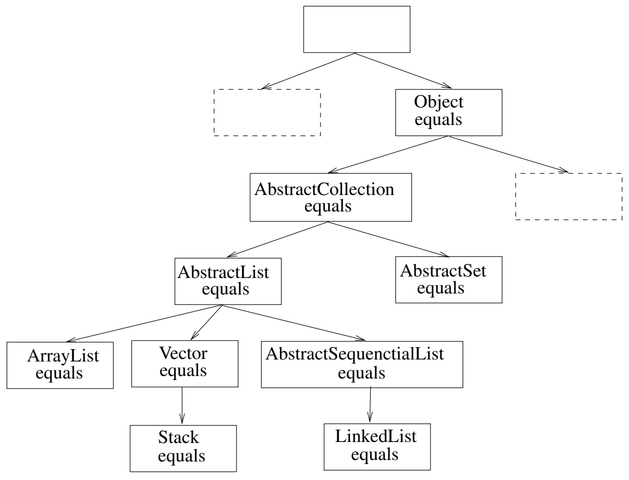


Figure 2. A partial call graph for the `equals` function in the `AbstractList` implementation.

to compute the distance estimate from the start to the end of the `equals` method we have to evaluate the cost of moving through the method calls in Figure. 1. The list iterator operations (lines 8-12 and 15) and the call to `equals` on the objects from both lists (line 13) are dynamic method invocations whose targets cannot be determined statically. A very small portion of the call graph with the class hierarchy for the method in Figure. 1 is shown in Figure. 2. The call graph shows that even for a single method call, there may be a large number of possible functions that we need to evaluate for computing heuristic estimates.

Simply minimizing the distance estimates across all the methods implemented by the sub-classes for a particular polymorphic method call is not computationally feasible. Our tests show that even for medium-sized programs such an analysis does not complete within a time bound of one hour. The heuristic estimates computed using such a brute force approach also tend to be inaccurate because at every program location it simply computes a lower-bound across all implementations. For programs with a large number of types the inaccurate estimates degenerate essentially into random estimates.

4. Polymorphic Distance Heuristic

In this section we present a new polymorphic distance estimate (PFSM) that performs an interprocedural static analysis to conservatively compute distance estimates with partial context information for targets of dynamic method invocations that are not statically resolved with a type analysis (unresolved polymorphic methods). It then dynamically computes the distance estimates on demand when the type of polymorphic methods are resolved during model checking. In other words, without completely analyzing all subtypes it lower-bounds distance estimates and computes the estimate on demand as type information is discovered at runtime.

4.1 Static analysis phase

An abstract model of the program is created to compute initial distance estimates. The model ignores all data values of the program and focuses only on control flow. The abstract model combines a control flow graph for each procedure in the program and a call graph that represents the call hierarchy of various procedures. The control flow graphs and the call graph denote the control flow of the program at an intra and inter procedural level respectively. The distance estimates between instructions in a procedure are computed as a lower-bound in the presence of branching and iterative constructs.

We algorithmically construct the abstract model and compute the distances between instructions in a method. The algorithm uses a reverse invocation order to estimate the cost of moving through method calls if the type of the callee can be statically determined. The analysis, however, does not step into methods whose type cannot be statically resolved after a rapid type analysis. In such cases a conservative estimate of two (one to call the function and another for the return edge) is assigned as the cost of moving through the corresponding call site to its immediate successor in the analysis. The conservative estimates are superseded by the distance estimates dynamically computed on demand as the type of methods is resolved during the model checking run.

The pseudo-code for the static analysis phase of computing the distance estimate values is presented in Figure. 3. The tuple, $\langle N, E, n_{start}, n_{end}\rangle$, is a control flow graph (CFG) where N is a set of abstract nodes labeled with unique program location identifiers, $E \subseteq N \times N$ is the set of edges, $n_{start} \in N$ is the start node, and n_{end} is the end node in the CFG. The variable, L, is a matrix of values that holds the distance estimates between instructions in each method. The $Explored$ variable is a map used to memoize the distance matrices for the different CFGs so that each method in the program is evaluated only once. The $Visited$ set is used to detect cycles in the control flow of a particular method. The function is_call_site takes as input a node in the CFG and returns true if the node represents a call site in the program. The has_resolved_type function takes as input a node that is a call site and returns true if the type of the target method (callee) is statically resolved after the rapid type analysis; the function get_target_CFG returns the CFG of the target method given a call site. Finally, the succ function returns a set of the immediate successors of a node n in the CFG, $\text{succ}(n) = \{n' \in N | (n, n') \in E\}$.

The polymorphic_distance_heuristic function is invoked by the `main` method to statically compute distance estimates as shown in Figure. 3 (lines 1-4). The function invokes the compute_estimates function with the CFG of the `main` method (lines 2-3). The compute_estimates function initializes a distance matrix $L : |N| \times |N|$ where the entries along the diagonal are set to zero while all other entries are set to ∞ (line 6). Next, on line 7 of Figure. 3, the analyze_function is called with the start node of the CFG, n_{start}, and the corresponding distance matrix, L, to initialize the edge costs between the nodes in the CFG.

The analyze_function uses a depth-first search traversal (lines 19-21) to update edge costs in L. For all nodes that are not call sites, the distance between the node and its immediate successor, d_{succ}, is set to one (line 18). When we encounter a call site during the traversal whose target method cannot be statically resolved then we conservatively set the cost of moving from the call site to its immediate successor node as two (lines 15-16). In essence, we do not evaluate any methods whose type cannot be statically resolved. If the type of the method can be resolved statically we update the cost between the call site and its successor by computing the distance estimate of moving through the target method (lines 12-14). After all the edge costs are updated in the distance matrix, L, the matrix is returned (line 22). At this point the analysis resumes on line 8 of the compute_estimates function where an all-pairs shortest path analysis is performed on the distance matrix. Finally the matrix is added to the $Explored$ map with its corresponding CFG (lines 8-9).

To compute the cost of moving through the target method of a call site, we invoke the get_distance_to_end function with the CFG of the target method and its corresponding call site (line 14). A simple check of whether the call site is also part of the target

```
procedure polymorphic_distance_heuristic($main$)
 1: /* N is set of nodes, E is the set of edges, $n_{start}$ is the start node, and $n_{end}$ is the end
    node in the CFG */
 2: $\langle N, E, n_{start}, n_{end} \rangle$ := get_CFG($main$)
 3: compute_estimates($\langle N, E, n_{start}, n_{end} \rangle$)
 4:
procedure compute_estimates($\langle N, E, n_{start}, n_{end} \rangle$)
 5: /* Entries along the diagonal are 0 while others are $\infty$ */
 6: $L : |N| \times |N| \rightarrow \mathbb{N} \cup \{\infty\}$
 7: $L$ := analyze_function($n_{start}, L, \emptyset$)
 8: $L$ := compute_all_pairs_shortest_distance($L$)
 9: $Explored$.add($\langle N, E, n_{start}, n_{end} \rangle, L$)
10:
procedure analyze_function($n, L, Visited$)
11: if is_call_site($n$) then
12:     if has_resolved_type($n$) then
13:         $\langle N', E', n'_{start}, n'_{end} \rangle$ := get_target_CFG($n$)
14:         $d_{succ}$ := get_distance_to_end($\langle N', E', n'_{start}, n'_{end} \rangle, n$)
15:     else
16:         $d_{succ} := 2$ /* Conservative estimate */
17: else
18:     $d_{succ} := 1$ /* Instructions other than call sites */
19: for each $n' \in$ succ($n$) and $n' \notin Visited$ do
20:     $L(n, n') := d_{succ}$; $Visited := Visited \cup \{n'\}$
21:     $L$ := analyze_function($n', L, Visited$)
22: return $L$
23:
procedure get_distance_to_end($\langle N, E, n_{start}, n_{end} \rangle, n_c$)
24: if $n_c \in N$ then
25:     return 2 /* Recursive call */
26: if $\neg Explored$.contains($\langle N, E, n_{start}, n_{end} \rangle$) then
27:     compute_estimates($\langle N, E, n_{start}, n_{end} \rangle$)
28: $L := Explored$.get_element($\langle N, E, n_{start}, n_{end} \rangle$)
29: return $L(n_{start}, n_{end})$
```

Figure 3. Pseudocode for computing distance estimates statically.

CFG reveals a recursive method call, in which case a conservative estimate of two is returned. In non-recursive method calls, if the target method is not found in the $Explored$ set, then we step into the target method by calling the compute_estimates function with the CFG of the target method (line 27). When the execution flow returns on line 28 of Figure. 3, we get the corresponding distance matrix, L, for the target method. The shortest distance from the start node to the end node in the distance matrix is returned as the cost of moving from the call site to its immediate successor on line 14 of the analyze_function.

4.2 Guided Search

The heuristic computed on the abstract model is used to intelligently rank the concrete states generated at points of thread non-determinism during model checking. A concrete state, s, contains a set of unique thread identifiers, a program location and stack for each thread and a heap. For each successor, s', of s the PFSM distance estimate from the current program location of the recently executed thread in s' to the specified target location is assigned as the heuristic rank of the s'. Intuitively, the PFSM heuristic drives certain threads toward the target locations specified by the user or generated using static analysis techniques.

4.3 Dynamic heuristic computation

The abstract model consisting of control flow graphs and the call graph of the program is refined when type information is discovered during model checking. As the refinement step we compute the distance estimates between the instructions in the control flow graph of a procedure whose alias information is discovered during the model checking run. The final heuristic value is computed on the abstract model along a sequence of call sites across the different control flow graphs from the current location to the target location.

We algorithmically show the heuristic computation in the dynamic analysis phase of the PFSM heuristic. The algorithm traverses the call graph in a depth-first manner to implicitly construct call traces between the current location and the target location in the forward direction. It uses the type information in the state generated during model checking to compute the distance estimates on demand along a particular call trace by using correct alias information to resolve types. The algorithm uses a branch and bound technique to restrict the number of call traces that need to be evaluated. The algorithm minimizes the distance estimate among all the call traces that lead from the current location to the target location. If the target location is not reachable in the forward direction then we look up the return point of the current function in the runtime stack extracted from the state generated during model checking as described in (Rungta and Mercer 2005). Next, if the target location is reachable from the return point we return the sum of the cost of moving to the end of the current function plus the distance estimate from the return point to the target location as the heuristic estimate; otherwise we keep unrolling the stack and repeat the above process.

The pseudocode for the dynamic phase of the algorithm is shown in Figure. 4. The get_forward_distance_estimate function in Figure. 4 takes as input the current program location of the most recently executed thread in the concrete state ($curLoc$) and the target location ($targetLoc$) to compute the distance estimate between them in the forward direction. The get_function_containing returns the CFG which contains the current program location (line 1). If the CFG containing the current location has not been previously analyzed (line 2) then we know that the type of a polymorphic method

```
procedure get_forward_distance_estimate($curLoc$, $targetLoc$)
 1: $\langle N, E, n_{start}, n_{end}\rangle$ := get_function_containing($curLoc$)
 2: if $\neg Explored$.contains($\langle N, E, n_{start}, n_{end}\rangle$) then
 3:     compute_estimates($\langle N, E, n_{start}, n_{end}\rangle$)
 4: if $targetLoc \in N$ then
 5:     return get_distance($curLoc$, $targetLoc$)
 6: return get_estimate(get_CFG_node($curLoc$), get_CFG_node($targetLoc$))
 7:
procedure get_estimate($n, n_t$)
 8: $hVal := \infty$
 9: for each $n' \in$ call_sites(get_function_containing($n$)) do
10:     if not_related($n', n_t$) then
11:         continue
12:     $d :=$ get_distance($n, n'$)
13:     if $d < hVal$ then
14:         $hVal :=$ min(compute_dynamic_estimate($n', n_t, d, hVal$), $hVal$)
15: return $hVal$
16:
procedure compute_dynamic_estimate($n_c, n_t, d, hVal$)
17: if $n_t \in$ target_CFG_nodes($n_c$) then
18:     $hVal' := d +$ get_distance_from_start_to_node($n_t$) $+ 1$
19:     return $hVal'$
20: else
21:     /* $CGR \subseteq X_c \times X_c$ where $X_c$ is the set of all call sites in the program. */
22:     for each $n'_c \in CGR(n_c)$ do
23:         if not_related($n'_c, n_t$) then
24:             continue
25:         $d' := d +$ get_distance_from_start_to_node($n'_c$) $+ 1$
26:         if $d' < hVal$ then
27:             $hVal :=$ min(compute_dynamic_estimate($n'_c, n_t, d', hVal$), $hVal$)
28:     return $hVal$
29:
```

Figure 4. Pseudocode for computing the distance heuristic during runtime

is now resolved. At this point we can compute the distance estimates between the instructions in the method (line 3) by calling the compute_estimates function in Figure. 3. Next, if the target node is contained within the same CFG as the current node (line 4) then the algorithm returns the value obtained from the get_distance function. The get_distance function returns the shortest distance between the two nodes in the same CFG. Note that the distance between the two nodes in the CFG is computed using partial context information because the algorithm conservatively assigns the distance between a call site for an unresolved polymorphic type to its immediate successor as two in the CFG.

The get_estimate and compute_dynamic_estimate functions traverse the nodes in the call graph implicitly constructing the call traces from the current location to the target location in the forward direction to compute the heuristic value, $hVal$. The function uses a branch and bound algorithm in an attempt to restrict the number of call traces that need to be evaluated for computing $hVal$. The function call_sites (line 9) generates the set of nodes that represent call sites in the input CFG while the not_related (lines 10 and 23) function returns true if there does not exist a path between the input nodes, n' and n_t (line 10), in the forward direction on the call graph. The get_estimate and compute_dynamic_estimate functions compute the distances along call traces using a depth-first traversal of the call graph (lines 9-14 and 22-27 respectively) such that the target node is reachable in the forward direction along the call trace. Note that we detect loops in the call trace in our implementation and backtrack appropriately. The get_estimate function constructs the first part of the call trace. It gets the distance from the current location to a call site within its own method that leads to the target node (lines 10-12). The get_estimate function then calls compute_dynamic_estimate (line 14) to compute the distance estimate on the rest of the call trace.

The compute_dynamic_estimate function computes the distances through the different call sites in a call trace. It uses a call graph relation, $CGR \subseteq X_c \times X_c$, where X_c is the set of the call sites in the entire program, to build a path through the different call sites in the program (lines 22-27) to a target location. Intuitively, a call graph relation describes the edges between different nodes in a call graph. The algorithm maintains a running summary of the distance estimates between the call sites (line 25). The get_distance_from_start_to_node function takes as input a node (which in this case is a call site) and gets the CFG that contains the input node. If the $Explored$ set contains the CFG then the get_distance_from_start_to_node function returns the shortest distance from the start node of the CFG, n_{start}, to the call site; otherwise it returns a conservative estimate of two. This essentially computes the distance estimates between different call sites in the call trace. When the algorithm reaches a call site whose callee CFG contains the target node, the function returns the summation of the distances along the path in the call trace up to the target node as the heuristic value (lines 17-19). The heuristic value is computed as a lower-bound and is propagated along the different call paths to prune other call traces when the value along a path becomes greater than the current heuristic value.

Theorem 4.1. *The PFSM heuristic computes a lower-bound on the distance estimate, if there exists one or more sequences of call points, $\langle c_0, c_1, \ldots, c_k\rangle$, in the call graph through k methods, in the presence of unresolved polymorphic methods, that represent a path between the current location, l, and the target location, t, $d_{min}(l,t) := d_{min}(l, c_0) + \sum_{i=1 to\ k-1} d_{min}(\mathsf{start}(i), c_i) + d_{min}(\mathsf{start}(k), t)$, between l and t and minimizes across all call sequences.*

Proof. Assume that the algorithm does not lower-bound the distance estimate along a particular sequence of call points. There are two possible cases when computing the distance along a sequence of call points. (1) The type of the method containing a call point is resolved—either statically or dynamically. Here the algorithm performs an all-pairs shortest path analysis on the CFG of the method. The analysis returns values between the nodes that are a lower-bound on the actual distance in the presence of branching and looping constructs. (2) The type of the method containing the call point is not resolved. The algorithm assigns a lower-bound of two to account for moving from the start of the method to a call site and then moving to next call site. The summation of all the values as the algorithm moves along a particular call sequence is a lower-bound on the distance estimate between l and t which contradicts our assumption. □

Corollary 4.2. *The PFSM heuristic estimate is consistent.*

Proof. The proof follows the one described for the FSM distance heuristic in (Edelkamp and Mehler 2003). There are two possible cases: (1) The shortest path from s to t contains s'. Suppose the length of the shortest path from s to t is l then, by definition, $h(s,t) = h(s',t) - d(s,s')$ which satisfies $h(s,t) \leq h(s',t) + d(s,s')$. (2) The shortest path from s to t *does not* contain s'. Consider the path, $\pi = s \rightarrow s' \rightarrow \ldots \rightarrow t$, where $|\pi| \geq l + d(s,s')$ and l is the shortest path between s and t. Furthermore, $\pi' = s \rightarrow \ldots \rightarrow t$, which implies $|\pi'| \geq l$ and $h(s') \geq l$. Hence we have $h(s,t) \leq h(s',t) + d(s,s')$. □

Corollary 4.3. *The PFSM heuristic is admissible.*

Proof. By definition, a consistent heuristic is also admissible (Russell et al. 1995). From Corollary 1 we know that the PFSM heuristic is admissible.

Theorem 4.4. *The complexity of computing the PFSM distance heuristic in the forward direction is* $O(\sum_{i=0}^{rm} |N_i|^3 + |N_c + E_c|)$ *where* rm *is the number of methods with resolved types,* N_i *is set of nodes representing instructions in method i,* N_c *is the set of nodes in the call graph, and* E_c *is the set of edges in the call graph.*

Proof. The complexity of computing the PFSM distance heuristic in the forward direction is $O(\sum_{i=0}^{rm} |N_i|^3)$ for performing an all-pairs shortest path analysis on every method whose type has either been statically (line 8 in Figure. 3) or dynamically resolved. Note that when the type of a method is dynamically resolved, line 3 in Figure. 4 calls the function compute_estimates in Figure. 3 and then performs the all-pairs shortest path analysis on line 8 in Figure. 3. The complexity of the PFSM distance heuristic is also linear in the number of nodes and edges in the call graph as it computes a lower-bound on the distance estimates across the different call sequences between the current and target location (lines 9-14 and 22-27 in Figure. 4). In the worst case, the algorithm explores the entire call-graph in a depth-first manner; in general, however, propagating the lower-bound across the different call sequences is successful in pruning a large number of call sequences that do not need to be explored. □

In contrast, the FSM distance heuristic, minimizes over all possible implementing sub-types of a particular method and has a complexity of $O(X^3)$, where X is the number of total instructions in the program $X = \sum_{1 \leq i \leq m} |N_i|$, and m is the total number of methods in the program; however, X is very large for most programs of interest. The PFSM heuristic is more computationally effective even though both heuristics belong to the same complexity class.

As the model checking run progresses, PFSM distance heuristic estimate, h_p, is a tighter lower-bound compared to the FSM distance heuristic estimate, h_f, such that $h_f \leq h_p$. The FSM distance heuristic is a context-insensitive algorithm and under-approximates distance values by ignoring all calling context. As the type of one or more methods are resolved during the model checking run the PFSM distance heuristic computes distances along different methods based on correct alias information. The PFSM distance heuristic uses the context information on the runtime stack of the state in a manner similar to the e-FCA (Rungta and Mercer 2006). A more detailed example demonstrating the effects of calling context is shown in (Rungta and Mercer 2006).

4.4 Example of heuristic computation

We use the example in Figure. 5 to demonstrate how the heuristic values are computed. The class `X` in Figure. 5(a) is an abstract class with three methods: `aa`, `test`, and `bb`. The classes `Y` (Figure. 5(b)) and `Z` (Figure. 5(c)) inherit from the `X` class. In Figure. 5(a), the input to the `test` method is an object, x, of type `X`. On lines 7 and 8, methods `bb` and `aa` are invoked, respectively, on the current instance of `X` and the input parameter x. Statically we can determine that the call on line 7 of the `test` method invokes the `bb` method on lines $11 - 15$ in Figure. 5(a); however, `aa` is a dynamically invoked method and the target of the call on line 8 of Figure. 5(a) depends on the type of x. The overall calling structure of the program is shown in Figure. 5(d). The `test` method in `X` can call the `aa` method in either the `Y` or `Z` class. The `aa` method then calls the `cc` method in its respective class. In the example shown in Figure. 5, the goal is to drive the program execution to line 9 in the `cc` method of the `Y` class in Figure. 5(b). Recall that in medium to large programs evaluating all possible implementing subtypes is intractable.

Suppose for the program shown in Figure. 5, a `main` method calls `test` with different instances of `X` objects. During the static analysis phase, when we reach the `test` function in Figure. 5(a), the analysis accounts for the cost of moving through the *this*.`bb` method call on line 7 in Figure. 5(a); however, the analysis cannot statically resolve the type of x; thus, the static analysis does not evaluate either implementation of `aa` in the `Y` or `Z` class and assigns a conservative estimate of two to account for the cost of moving from line 8 to the end of the `test` method. At the end of the static analysis, the *Explored* set only contains the `test` and `bb` methods.

Let us consider two cases in the dynamic computation of the heuristic. In the first case, suppose the current location of the program is at line 6 in Figure. 5(a) and we want to compute a distance estimate to line 9 in Figure. 5(b). We first get all the call sites that are reachable from the current location such that there exists a path from the call site to the target location on the call graph and the call sites are in the same CFG as the current location. The only call site that satisfies the condition in Figure. 5 is x.aa(). We then call the get_estimate() function in Figure. 4 with the corresponding call site. The x.aa() call site can call the aa function in either the `X` class or the `Y` class. This maps to two entries in the call graph relation: Y.aa() $\rightarrow$ Y.cc() and Z.aa() $\rightarrow$ Z.cc(); however, the target location can only be reached from Y.cc() based on the calling hierarchy shown in Figure. 5(d). Since the distance estimates in the `aa` method of the `Y` class have not been computed on the CFG (as the method does not currently exist in the *Explored* set), a conservative cost of two is added along the call trace when moving from x.aa() to Y.cc(). Similarly, a conservative estimate of two is added for the cost of moving to the `cc` method in the `Y` class and reaching the target at line 9 because the `cc` method does not exist in the *Explored* set. A final heuristic estimate of four is returned for the example.

```
1:  public abstract class X{
2:
3:    public abstract void aa();
4:
5:    public void test(X x){
6:      i := 0;
7:      this.bb();
8:      x.aa();
9:    }
10:
11:   public void bb(){
12:     this.val := 10;
13:     this.otherVal := 11
14:   }
15: }
```

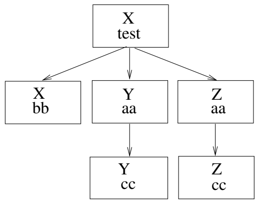
(a)

```
1:  class Y extends X{
2:
3:    public void aa(){
4:      this.cc();
5:    }
6:
7:    public void cc(){
8:      if(...) then
9:          throw RuntimeException()
10:   }
11: }
```

(b)

```
1:  class Z extends X{
2:
3:    public void aa(){
4:      this.cc();
5:    }
6:
7:    public void cc(){
8:    /* Local Instruction */
9:    }
10: }
```

(c)



(d)

Figure 5. An example program and its corresponding call graph to demonstrate the heuristic computation.(a) An abstract class, `X`, with an abstract method and implementations for two functions. (b) The `Y` class that inherits from the `X` class. (c) The `Z` class that inherits from the `X` class. (d) The call graph for the functions in `X`, `Y`, and `Z`.

In another example that demonstrates the dynamic computation of the heuristic, suppose the current location of the program is at line 4 in Figure. 5(b). The location implicitly resolves the type of the `aa` method in the `Y` class because the model checking search is at the method. At this point we run the static analysis algorithm (shown in Figure. 3) on the `aa` method in Figure. 5(b). Note that since the target of this.cc() is dynamically resolved the static analysis technique computes the cost of moving through the `cc` method at line 4 in Figure. 5(b). After refining the distance estimates, we return to the dynamic heuristic computation in Figure. 4. The analysis computes the distance estimates on the call trace $Y.\mathsf{aa}() \rightarrow Y.\mathsf{cc}()$ based on the shortest distances in the CFGs of the methods `aa` and `cc` in the `Y` class. The distance estimates in a CFG lower-bounds all values across the iterative and looping constructs.

5. Results

The experiments are conducted on machines with 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of guided search with various heuristics. Note that we break all heuristic ties randomly that enables us to overcome the benefits and limitations of a default search order in guided search (Rungta and Mercer 2007b). All the trials are time bounded at one hour. This is consistent with other empirical studies (Rungta and Mercer 2007b,a; Dwyer et al. 2006). Since each trial is completely independent of the other trials we use a super computing cluster of 618 nodes to distribute the trials on different nodes.[1] Even though the algorithm does not require parallel computation, using the super computing cluster allows us to quickly generate results. We use the Java Pathfinder (JPF) v4.1 model checker with partial order reduction turned on to conduct the experiments described in the paper. JPF model checks Java byte-code using a modified virtual machine.

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

The input to the guided search is the model and possible error locations. The possible error locations are be derived by user-specified reachability properties, can be generated by static analysis tools, or generated from dynamic analysis tools (Artho and Biere 2001; Hovemeyer and Pugh 2004; Havelund 2000). For example, static analysis tools report program locations where lock acquisitions by unique threads *may* lead to a deadlock. These tools, however, cannot state the feasibility of the deadlock. We use the technique described in (Rungta and Mercer 2008) to generate a small sequence of program locations that are relevant to checking the reachability of the possible error locations.

We use five unique multi-threaded Java programs in this study to evaluate the effectiveness of the PFSM heuristic. Three programs are from the benchmark suite of multi-threaded Java programs gathered from academia, IBM, and classical concurrency errors described in literature (Dwyer et al. 2006). 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 search is unable to find errors in the models (Rungta and Mercer 2007a). These models also exhibit different concurrency error patterns described by Farchi *et. al* in (Farchi et al. 2003). The `AbsList` and the `AryList` are programs that use the JDK 1.4 library in accordance with the documentation. We use Jlint on the `AbsList` and `AryList` models to automatically generate warnings on possible concurrency errors and then manually generate the input sequences as described in (Rungta and Mercer 2008). The name, type of model, number of locations in the input sequence, and source lines of code (SLOC) for the models are as follows:

EXPLORED STATES

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	15249	109259	409156	3279	30193	178653	209	213	217
Twostage(8,1)	23025	204790	603629	5956	46259	281132	246	251	255
Twostage(10,1)	36056	364859	1216340	14232	156697	1302040	329	335	340
Wronglock(1,10)	58	7064	49100	75	196	2362	367	3781	15923
Reorder(5,1)	1803	6006	12529	912	2562	5765	106	109	112
Reorder(8,1)	10155	34193	98683	5422	24022	96681	193	197	202
Reorder(10,1)	24890	80160	343429	6785	65506	149916	266	272	277
AryList(1,10)	3652	15972	63206	-	-	-	846	5216	50904
AbsList(1,10)	10497302	10497302	10497302	-	-	-	982	982	982

TIME IN SECONDS

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	4.52	40.14	124.17	33.49	39.11	65.93	0.37	0.42	2.57
Twostage(8,1)	6.70	76.24	184.82	34.65	41.87	83.45	0.39	0.41	0.49
Twostage(10,1)	10.89	132.08	318.93	36.35	59.90	242.78	0.43	0.46	0.52
Wronglock(1,10)	0.22	2.85	12.46	10.25	10.70	12.49	0.48	1.66	4.24
Reorder(5,1)	1.19	2.34	4.08	12.78	13.37	14.41	0.28	0.31	0.67
Reorder(8,1)	3.59	9.70	34.72	13.90	17.84	34.02	0.34	0.39	0.54
Reorder(10,1)	6.81	25.62	97.33	14.31	26.30	41.99	0.37	0.41	0.45
AryList(1,10)	2.12	7.95	26.11	-	-	-	12.21	13.60	22.56
AbsList(1,10)	2585.79	2585.79	2585.79	-	-	-	4.85	4.92	5.92

MEMORY IN MB

Subject	Random Heuristic			FSM Heuristic			PFSM Heuristic		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Twostage(7,1)	219	972	2090	922	1325	2219	160	182	203
Twostage(8,1)	352	1415	2541	961	1462	2411	163	178	203
Twostage(10,1)	508	2038	3902	1033	1886	3227	163	181	203
Wronglock(1,10)	18	117	374	204	434	693	77	114	187
Reorder(5,1)	50	89	166	185	348	590	160	179	203
Reorder(8,1)	159	387	848	214	571	1168	160	173	203
Reorder(10,1)	279	851	1856	362	1032	1453	163	179	203
AryList(1,10)	136	275	572	-	-	-	280	318	391
AbsList(1,10)	6154	6154	6154	-	-	-	165	193	256

Table 1. Comparing the performance of various heuristics.

- **TwoStage**: Benchmark, Num of locs: 2, Null Pointer Exception, SLOC: 52
- **Reorder**: Benchmark, Num of locs: 2, Null Pointer Exception, SLOC: 44
- **Wronglock**: Benchmark, Num of locs: 3, Deadlock due to inconsistent locking. SLOC: 38
- **AbsList**: Real, Num of locs: 6, Race-condition in the AbstractList class using the synchronized Vector sub-class. SLOC: 7267
- **AryList**: Real, Num of locs: 6, Race-condition in the ArrayList class using the synchronized List implementation. SLOC: 7169

Exhaustive search techniques like randomized depth-first search either struggle or fail to find an error in the models used in the empirical study. A more detailed comparison with a randomized depth-first search is shown in (Rungta and Mercer 2008).

We use a greedy depth-first search to guide the search. 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. We observe comparable results with a traditional greedy best-first search with a bounded queue. We use the distance heuristic to guide the search through each of the input locations generated using the technique in (Rungta and Mercer 2008) to mimic a test-like paradigm. The effects of varying the length of the sequence on the performance of the heuristic are also reported in (Rungta and Mercer 2008).

Only a portion of the frontier states are saved as backtrack points which turns the complete search into a partial search; however, our aim is to find a counter-example efficiently rather than to do an exhaustive proof or find an optimal counter-example. It is important to note that in medium to large programs, it is intractable to generate optimal counter-examples using an A^* search because it exhausts the memory resources very quickly.

In Table 1 we specifically compare the performance of the PFSM heuristic with the FSM and random heuristic while guiding the program execution through a small sequence of locations. The entries in Table 1 with "-" in the FSM heuristic columns indicate that the static analysis did not finish within the time bound of one hour.

The performance of the PFSM heuristic is dramatically better than the random and FSM heuristic. In the `Twostage(7,1)` model the PFSM heuristic generates a mere 213 states, on average, before error discovery while the random and FSM heuristic generate $109,259$ and $30,193$ states respectively, on average, in the error discovering trials. A similar improvement for the PFSM heuristic is noticed in the total time taken and memory used. In the `TwoStage(7,1)` model the PFSM only takes 0.42 seconds on average for error discovery, in contrast, the random heuristic takes 40.14 seconds while the FSM heuristic takes 39.11 seconds. In some models such as `Reorder(5,1)` and `Wronglock(1,10)` where the magnitude of states generated is small, the memory usage of the random heuristic is lower than the PFSM heuristic because it does not incur the additional heuristic computation cost. Note that the variance in the results using the FSM and PFSM heuristics is caused because we break all ties in heuristic values randomly.

6. Discussion

Recent work and our experience in testing and verifying multi-threaded programs show that only a small number of perturbations to certain global or shared variables are required to find a particular error in the multi-threaded system. The key, however, lies in discovering and driving the program execution through these perturbations to elicit the error. Recent work uses the output of static analysis warnings to generate a sequence of interesting programs relevant for verifying the feasibility a particular static analysis warning. The sequence is small with large gaps between each location. We rely on the distance estimate heuristic presented in this paper to guide the program execution toward the locations in the sequence. In essence, the distance heuristic drives certain threads toward specific program locations without manual intervention that is required in the other heuristics such as the prefer-thread heuristic (Groce and Visser 2002). This allows us to scale to realistic benchmarks and discover errors after exploring only a few hundred states.

The heuristic lower-bounds the values across loops and recursive function calls. If the loops and recursive function calls operate solely on local variables the dynamic partial order reduction allows us to process a series of instructions as a single transaction; however, if they operate on global variables then the distance heuristic is sufficient to drive a particular thread through a loop until it exits the loop and moves toward the location of interest.

7. Related Work

Heuristics have extensively been used for error detection in program and system verification. Hamming distance heuristics presented by Yang and Dill use the explicit state representation to estimate a bit-wise distance between the current state and an error state (Yang and Dill 1998). Edelkamp, Lafuente, and Leue implemented a property based heuristic search which considers the minimum number of changes required to the variables in the property in order for the property to be violated. This information is used to estimate the distance to the error (Edelkamp et al. 2001b). This heuristic was refined using a Bayesian meta-heuristic by Seppi, Jones and Lamborn (Seppi et al. 2006). The heuristics in the FLAVERS tool uses the structure of the property to guide the search (Tan et al. 2004). The property based heuristics are not very effective for finding errors in object-oriented multi-threaded Java and C# programs because a large number of operations in the program do not directly affect the property being verified.

The heuristics in (Groce and Visser 2002) exploit the properties of Java programs to find concurrency errors. A variety of domain specific heuristics are proposed to find different concurrency errors; for example, the `most-blocked` heuristic prefers states with a greater number of blocked threads in order to find deadlocks, while the `prefer-thread` heuristic allows the user to specify a set of threads whose execution will be preferred over the other threads. The `prefer-thread` heuristic is effective for error discovery in certain programs where exhaustive model checking techniques fail to find an error as shown in (Rungta and Mercer 2007b); however, a considerable manual effort is expended while configuring the correct parameters required for error discovery in the models.

Distance heuristics are structural heuristics that have been extensively evaluated in this work, (Edelkamp and Mehler 2003; Cobleigh et al. 2001; Rungta and Mercer 2005, 2006). In essence, the success of these heuristics lie in the fact that they exploit the structure of the program to drive the program execution toward a set of interesting locations either specified by the user or generated using static/dynamic analysis techniques. The combination of distance estimate heuristics with the meta heuristic that guides the search along a sequence of program locations is considerably successful in detecting errors in multi-threaded programs (Rungta and Mercer 2008). The sequence of program locations generated manually essentially represents an abstract trace through the program while the distance heuristic enables us to find a corresponding feasible concrete execution path.

There are other related guided techniques that use different abstract traces and guidance strategies for either error discovery or optimal counter-example generation. The trail directed model checking generates a concrete counter-example to the error state using a depth-first search and uses the counter-example produced to guide the search toward an optimal counter-example (Edelkamp et al. 2001a). The goal of the distance heuristic presented in this paper with the meta heuristic, (Rungta and Mercer 2008), is to *discover errors* in programs where exhaustive search techniques such as depth-first search fail.

The deterministic execution technique uses a sequence of relevant data input to execute branch conditions, thread schedules, and method sequences generated manually by a tester to check whether an error exists in concurrent Java programs (Harvey and Strooper 2001). This, however, requires a significant amount of manual effort. The distance heuristic can intelligently rank thread schedules to drive certain threads along a small sequence of interesting locations.

Using an abstraction to guide a concrete execution of the system has also been explored in hardware verification. One approach generates a trace on an abstract model created using a set of initial boolean variables to represent the transition relation (Nanshi and Somenzi 2006). Next, a guided simulation using pseudo random number vectors guides the simulation of the concrete model to find a concrete counter-example. It refines the abstraction by adding more boolean variables. This work, however, is limited to verifying circuit designs and boolean programs. Other approaches use different abstraction and guidance techniques but, again, are limited to boolean programs (Paula and Hu 2007).

Concolic testing executes the program with random concrete values in conjunction with symbolic execution to collect the path constraints over input data values (Sen et al. 2005; Sen and Agha 2007) in order to systematically test programs. A new approach guides the concolic test along different branches to obtain better branch coverage (Burnim and Sen 2008). The guidance strategy is able to be achieve better branch coverage compared to traditional systematic techniques. This shows guidance strategies can also be combined with concolic test. An interesting avenue of future work would be to use guidance strategies in concolic testing to find specific errors.

8. Conclusions and Future Work

In this work we present a distance heuristic function that computes estimates in programs with unresolved polymorphic methods. The PFSM heuristic performs an interprocedural static analysis to conservatively compute distances estimates and, then, dynamically computes the distance estimates on demand after the types of polymorphic methods are resolved at transaction boundaries during model checking. The empirical analysis shows that the PFSM heuristic outperforms the FSM distance heuristic that ignores the calling context information and the baseline random heuristic. In future work we want to study and evaluate the trade-off between the accuracy in the heuristic estimate and the performance in the heuristic computation in how it affects the effectiveness of the guided search. For example, to further improve the accuracy of the distance estimate we can propagate the types—extracted from the state—along the program, as far as possible. A def-use analysis could be used to detect how far we can propagate the values in the program.

References

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.

D. F. Bacon and P. F. Sweeney. Fast static analysis of c++ virtual function calls. In *OOPSLA '96: Proceedings of the 11th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 324–341, New York, NY, USA, 1996. ACM. ISBN 0-89791-788-X. doi: http://doi.acm.org/10.1145/236337.236371.

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.

J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, pages 443–446. IEEE, 2008.

J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *International Conference on Software Engineering*, pages 37–46, 2001. URL `citeseer.ist.psu.edu/cobleigh01right.html`.

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. ISBN 1-59593-468-5. doi: http://doi.acm.org/10.1145/1181775.1181787.

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.

S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In S. D. Stoller and W. Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 55. Elsevier Science Publishers, 2001a.

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, 2001b.

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. ISBN 0-7695-1926-1.

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.

C. Harvey and P. Strooper. Testing Java monitors through deterministic execution. In *ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering*, page 61, Washington, DC, USA, 2001. IEEE Computer Society.

K. Havelund. Using runtime analysis to guide model checking of java programs. In *Proceedings of the 7th International SPIN Workshop on SPIN Model Checking and Software Verification*, pages 245–264, London, UK, 2000. Springer-Verlag. ISBN 3-540-41030-9.

T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In T. Ball and S.K. Rajamani, editors, *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, May 2003.

G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

D. Hovemeyer and W. Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, 2004. ISSN 0362-1340. doi: http://doi.acm.org/10.1145/1052883.1052895.

K. Nanshi and F. Somenzi. Guiding simulation with increasingly refined abstract traces. In *DAC '06: Proceedings of the 43rd annual conference on Design automation*, pages 737–742, New York, NY, USA, 2006. ACM. ISBN 1-59593-381-6. doi: http://doi.acm.org/10.1145/1146909.1147097.

F. M. De Paula and A. J. Hu. An effective guidance strategy for abstraction-guided simulation. In *Design Automation Conference (DAC)*, pages 63–68, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-627-1. doi: http://doi.acm.org/10.1145/1278480.1278498.

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.

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.

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. ISBN 0-7695-2707-8. doi: http://dx.doi.org/10.1109/FMCAD.2006.5.

N. Rungta and E. G. Mercer. A meta heuristic for effectively detecting concurrency errors. In *Haifa Verification Conference, To Appear.*, Haifa, Israel, 2008.

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 2007a.

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 2007b. Springer–Verlag.

S.J. Russell, P. Norvig, J.F. Canny, J. Malik, and D.D. Edwards. *Artificial intelligence: a modern approach.* Prentice Hall Englewood Cliffs, NJ, 1995.

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. ISBN 978-3-540-70888-9.

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*, pages 263–272, New York, NY, USA, 2005. ACM. ISBN 1-59593-014-0. doi: http://doi.acm.org/10.1145/1081706.1081750.

K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informaticae*, 70(1-2):111–126, 2006.

J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in flavers. In *SIGSOFT '04/FSE-12: Proceedings of the 12th ACM SIGSOFT Twelfth International Symposium on Foundations of Software Engineering*, pages 201–210, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-855-5. doi: http://doi.acm.org/10.1145/1029894.1029922.

W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *35th Design Automation Conference (DAC98)*, pages 599–604, 1998.