A distance heuristic for polymorphic programs

Neha Rungta and Eric G Mercer

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

Abstract. Recent work for testing multi-threaded programs uses a sequence of locations (partial error trace) manually generated from the output of static analysis tools to guide the program execution in a model checker along a depth-first search path to verify the feasibility of the error. This guided test technique uses a two-level ranking scheme where states are first ranked based on the number of locations in the partial error trace already observed and then ranked based on the perceived cost to reach the next location in the error trace. The effectiveness of the test critically relies on the ability of the secondary heuristic to compute the perceived cost to the next location in the sequence.
Distance estimate heuristics lend themselves naturally to guide the test toward the next location in the sequence. These heuristics compute the distance estimate between two program locations on the control flow representation of the program with varying degrees of context information. Current distance heuristic estimates, however, are unable to compute accurate heuristic estimates in the presence of unresolved polymorphic methods. In this paper we describe a polymorphic distance heuristic that uses a rapid type analysis to reduce the number of unresolved polymorphic method types in the program. Then it performs an interprocedural static analysis to conservatively compute distances estimates. Finally, to compute the heuristic value it dynamically refines the distance estimates when the types of polymorphic methods are resolved at transaction boundaries during model checking. In an empirical analysis, we show that the polymorphic distance heuristic in a guided test setting outperforms the other heuristics in a guided test setting.

1 Introduction

The ubiquity of multi-core processors is creating a paradigm shift from inherently sequential to highly concurrent and parallel systems. In May 2007, Microsoft Chief Research and Strategy Officer, Craig Mundie described the need to "reliably construct large-scale applications that are distributed, highly concurrent, and able to utilize all this computing power [1]." The lack of effective tools and techniques to detect concurrency errors is proving to be a deterrent for programmers to develop concurrent programs. It also causes programmers to use concurrency conservatively leading to decreased performance. The trend toward parallelism and concurrency motivates a need to develop effective verification and testing techniques for concurrent programs.

The current state of the art testing and verification techniques are insufficient for testing and verifying the presence of concurrency errors in multi-threaded programs. The techniques lie on various points on a spectrum of precision and scalability. On one end of the spectrum, static analysis techniques [2–7] are imprecise but can scale to large systems. The analysis techniques mostly ignore the semantics of the program and analyze the source code to report warnings about possible errors. The errors reported by static analysis techniques have to be manually verified which is difficult, tedious, and sometimes not possible. At the other end of the spectrum, model checking techniques, [8–11], are precise and sound but do not scale to large systems.

Model checking techniques exhaustively enumerate all possible behaviors of the system to verify the presence as well as the absence of errors in programs. The exhaustive nature of model checking leads to huge state space explosion making it infeasible in practical applications. A large body of literature has been devoted to overcoming the state space explosion problem in sequential programs; for example, symbolic execution [12] and counter-example guided abstraction refinement [10, 13]. In contrast, for concurrent systems, only a few techniques such as partial order reduction, [14, 15], and thread symmetry reduction [16], have been partially successful in reducing the number of thread schedules and state configurations explored in model checking to show correctness. While these techniques move the limit on the size of the programs we can analyze, they do not scale to practical applications in concurrent programs; thus, there is a need for a sound and scalable technique that can effectively test real world concurrent programs.

Recent work on guided test, [17], attempts to bridge the gap between precision and scalability to verify the presence of concurrency errors in multi-threaded programs. A guided test directs the program execution through a small sequence of program locations (partial error trace) manually generated from warnings reported by static analysis tools such as FindBugs [6] and Jlint [7]. The sequence may or may not be feasible in any valid execution of the program. It serves as a starting point for verifying a possible error. The guided test technique directs the program execution in a greedy depth-first manner using a two-level ranking scheme. The two-level ranking is based on the following criteria: (1) the number of locations in the partial error trace already observed (we will refer to this as the sequence-based meta-heuristic); and (2) the perceived cost to reach the next location in the error trace. States that have observed a higher number of locations from the sequence are ranked as more *interesting* compared to states that have observed fewer locations in the sequence. A secondary heuristic estimate based on the proximity to the next location in the sequence is used to rank the states that have observed the same number of locations from the sequence. The guided test uses the two-level ranking to pick the best successor of a given state in a depth-first search. Note, however, that the ranking scheme described above can also be used in a greedy best-first search or other prioritized search orders. The results in [17] suggest that guided test is successful in quickly verifying errors in

the JDK 1.4 concurrent library using sequences generated from static analysis warnings and has the potential to scale to practical concurrent applications.

The success of guided test using the two-level heuristic is dramatically affected by the accuracy of the secondary heuristic in guiding the test toward the next location [17]. Distance heuristics, [18–20], lend themselves naturally to guide the test toward the next location in the sequence. The current distance heuristics are effective for error discovery in C programs with resolved function pointers [20, 21] and Java programs with resolved method types [18] when used in a greedy best-first search without the sequence of locations. The distance estimate heuristics, however, are unable to statically compute the cost of calling unresolved polymorphic functions in object-oriented languages such as Java, C++, and C#.

The work in [17] describes the guided test and the two-level ranking scheme; while abstracting away the details about the secondary distance heuristic used to compute distance estimates in polymorphic programs. This paper presents the algorithm and the technique to compute the distance estimates for polymorphic programs that is effective in a guided test setting.

The polymorphic distance heuristic we present in this work uses a rapid type analysis to reduce the number of unresolved polymorphic method types in the program, it then performs an interprocedural static analysis to conservatively compute distance estimates, and finally, it dynamically refines the distance estimates when the types of polymorphic methods are resolved at transaction boundaries during model checking. For example, in the Java Pathfinder model checker, [9], the transaction boundaries are computed using a dynamic partial order reduction technique. The transaction boundary is a program location where the model checker considers the different non-deterministic choices. In this work we only consider non-determinism arising from thread-schedules in the system models.

We further present an empirical analysis to demonstrate the effectiveness of the polymorphic distance heuristic in guided test for error discovery. We present an analysis to show that a guided test is more effective in error discovery compared to a greedy best-first search when using a two-level ranking scheme. We, also, summarize the results from [17] that demonstrate that the polymorphic distance heuristic outperforms other heuristics in a guided test setting.

2 Motivation

The work in [17] detects an error in the JDK 1.4 library method shown in Fig. 1. We detect a concurrent modification exception in the `equals` function of the `AbstractList` class when using two instances of the synchronized, `Vector` sub-class implementation (e.g., `AbstractList` l_1 = `new Vector`()). We detect the error with the guided test technique using the polymorphic distance estimate heuristic presented in this paper. The effectiveness of the guided test critically relies on the accuracy of the distance estimates computed by the polymorphic distance estimate heuristic presented in this paper.

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

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

Consider the example shown in Fig. 1, if we want to compute the distance estimate from the start to the end of the `equals` method in Fig. 1, we have to evaluate the cost of moving through the method calls in Fig. 1. The list iterator operations (lines 8-12 and 15) and the call to `equals` on the objects from both lists (line 13) are polymorphic methods. In the current static analysis technique, to compute distance heuristics we cannot resolve the type of the method call, o_1.equals(o_2) on line 13. A very small portion of the call graph with the class hierarchy for the method in Fig. 1 is shown in Fig. 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 because for certain polymorphic methods there are a number of sub-classes implementing the method.

A brute force approach of just 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 one hour time bound. The heuristic estimates computed using such a technique also tend to be inaccurate because at every program location the brute force approach simply computes a lower-bound across all implementations. For programs with a large number of types such inaccurate estimates degenerate essentially into random estimates.

We can use an extremely precise type analysis to resolve the method types before computing the distance estimates; however, the accuracy of the precise type analysis techniques causes a significant trade-off in the performance because now the computation resources such as time and memory are expended in the

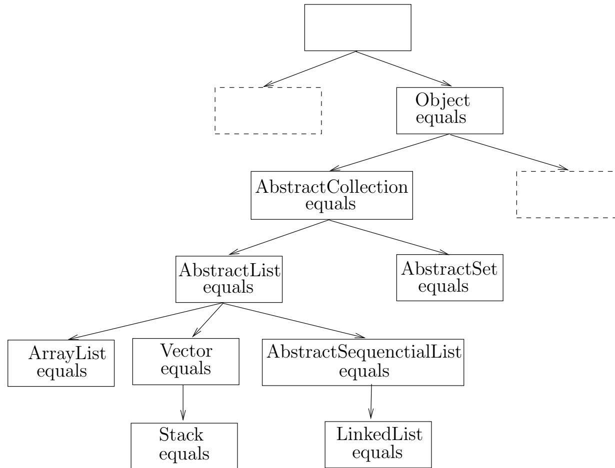


Fig. 2. A partial call graph for the `equals` function in the `AbstractList` implementation.

type analysis rather than state exploration. There is a need for a technique that computes the distance estimate in the presence of polymorphism while maintaining a balance among the accuracy of the heuristic, the effectiveness of the heuristic in guided search or test, and the performance overhead incurred in the heuristic computation.

3 Polymorphic Distance Heuristic

The e-FCA distance heuristic, [20], computes full context aware distance estimates in non-recursive C programs with resolved function pointers and improves on previous distance estimates based on the FSM heuristic function, [18], with no context and the EFSM heuristic function, [19], with partial context. The e-FCA distance estimate is computed based on the 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 (we also refer to this notion as 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 uses the information on the dynamic runtime stack, extracted from the state generated during model checking, to look up the return point of the current function. If the target location is reachable in the forward direction from the return point, 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 heuristic,

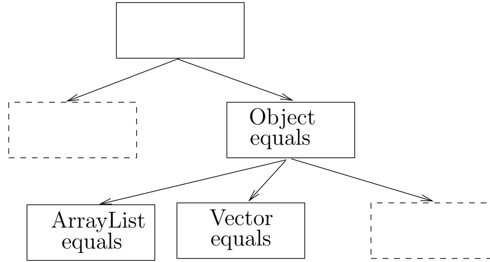


Fig. 3. Reduced call graph after performing the Rapid Type Analysis

however, cannot compute distance estimates in the forward direction with full context information in the presence of unresolved polymorphic methods without minimizing over all implementing sub-types, which is computationally infeasible.

In this section we present a new polymorphic distance estimate (PFSM) that uses a low-cost rapid type analysis to reduce the number of unresolved polymorphic method types in the program. After the type analysis, the heuristic performs an interprocedural static analysis to conservatively compute distance estimates with partial context information for unresolved polymorphic methods. It then refines the distance estimates, dynamically, when the types of polymorphic methods are resolved at transaction boundaries generated by a dynamic partial order reduction during model checking. In other words, without completely analyzing all subtypes it lower-bounds distance estimates and refines the bound as type information is discovered at runtime.

3.1 Type analysis

There are several type analysis algorithms proposed in literature to statically resolve types; however, a trade-off between precision and performance is observed in these algorithms. A higher analysis cost is associated with the more precise algorithms and vice-versa. The rapid type analysis, [22], is a low-cost, fast, and flow-insensitive type analysis technique that uses the information about instantiated classes to create a reduced set of executable methods in polymorphic programs. In this work we use the rapid type analysis to prune the number of methods that need to be evaluated while computing the distance estimate heuristic. Suppose in the program using the `AbstractList` equals function shown in Fig. 1, only the `Vector` and `ArrayList` classes are initialized. Given this scenario, the call graph after performing the rapid type analysis on the graph shown in Fig. 2 looks like the graph Fig. 3. This example demonstrates that a large number of polymorphic methods can be pruned by performing the low-cost rapid type analysis.

```
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: ⟨N, E, n_start, n_end⟩ := get_CFG(main)
 3: compute_estimates(⟨N, E, n_start, n_end⟩)
 4:
procedure compute_estimates(⟨N, E, n_start, n_end⟩)
 5: /* Entries along the diagonal are 0 while others are ∞ */
 6: L : |N| × |N| → ℕ ∪ {∞}
 7: L := analyze_function(n_start, L)
 8: L := compute_all_pairs_shortest_distance(L)
 9: Explored.add(⟨N, E, n_start, n_end⟩, L)
10:
procedure analyze_function(n, L)
11: if is_call_site(n) then
12:    if has_resolved_type(n) then
13:       ⟨N′, E′, n′_start, n′_end⟩ := get_target_CFG(n)
14:       d_succ := get_distance_to_end(⟨N′, E′, n′_start, n′_end⟩, n)
15:    else
16:       d_succ := 2 /* Conservative estimate */
17: else
18:    d_succ := 1 /* Instructions other than call sites */
19: for each n′ ∈ succ(n) do
20:    L(n, n′) := d_succ
21:    L := analyze_function(n′, L)
22: return L
23:
procedure get_distance_to_end(⟨N, E, n_start, n_end⟩, n_c)
24: if n_c ∈ N then
25:    return 2 /* Recursive call */
26: if ¬Explored.contains(⟨N, E, n_start, n_end⟩) then
27:    compute_estimates(⟨N, E, n_start, n_end⟩)
28: L := Explored.get_element(⟨N, E, n_start, n_end⟩)
29: return L(n_start, n_end)
```

Fig. 4. Pseudocode for the static analysis phase of computing the distance estimates in polymorphic programs.

3.2 Static analysis phase

The static analysis phase of the heuristic computes the distances between instructions in a method using 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 and 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 pseudo-code for the static analysis phase of computing the distance estimate values is presented in Fig. 4. The tuple, $\langle N, E, n_{start}, n_{end}\rangle$, is a control flow graph (CFG) where N is a set of uniquely labeled nodes, $E \subseteq N \times N$ 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 function is_call_site takes as input a node in the CFG and returns true if the 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, $\mathsf{succ}(n) = \{n' \in N | (n, n') \in E\}$.

The distance estimate heuristic for polymorphic programs invokes the function, polymorphic_distance_heuristic, with the `main` method of the program, under test, to statically compute distance estimates as shown in Fig. 4 (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 Fig. 4, the analyze_function is called with the start node of the CFG, n_{start}, and the corresponding distance matrix, L, to store 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, if the type of the target method is not resolved, 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, and 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 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 Fig. 4, 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.

```
procedure get_forward_distance_estimate(curLoc, targetLoc)
 1: ⟨N, E, n_start, n_end⟩ := get_function_containing(curLoc)
 2: if ¬Explored.contains(⟨N, E, n_start, n_end⟩) then
 3:    compute_estimates(⟨N, E, n_start, n_end⟩)
 4: if targetLoc ∈ 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 := ∞
 9: for each n' ∈ 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 ∈ 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 ⊆ X_c × X_c where X_c is the set of all call sites in the program. */
22:    for each n'_c ∈ 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:
```

Fig. 5. Pseudocode for computing the distance heuristic during runtime

3.3 Dynamic heuristic computation

The algorithm first refines the distance estimates, when the types of polymorphic methods are resolved; next, it computes the distance estimates in the forward direction by constructing different call traces (sequences of method calls) that represent a path from the current location to the target location in the forward direction. If the target location is not reachable in the forward direction, same as the e-FCA heuristic [20], we look up the return point of the current function in the runtime stack extracted from the state generated during model checking. If the target location is reachable in the forward direction from the return point, the heuristic value is summation of cost of moving to the return point with the distance estimate in the forward direction from the return point. Otherwise, we keep unrolling the stack until we reach the top level function.

The pseudocode for the dynamic phase of the algorithm is shown in Fig. 5. The get_forward_distance_estimate function, in Fig. 5, takes as input the cur-

rent location (*curLoc*) of the program 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), we know that the type of a polymorphic method 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 Fig. 4. Next, if, the target node is contained within the same CFG as the current node (line 4) the algorithm returns the value obtained from the get_distance function. The get_distance 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. Also note that the get_forward_distance_estimate is only called when the model checker stops at transaction boundaries as determined by dynamic partial order reduction. As a consequence only some of the types are resolved and this saves the analysis cost on the other methods.

The get_estimate function and compute_dynamic_estimate, together, construct 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. 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 the 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. 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.

```
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: }
```

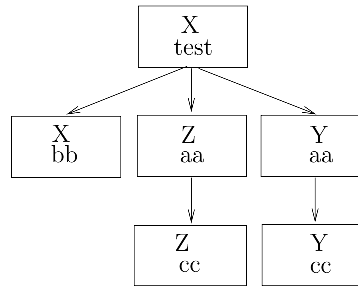
(a)

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

(b)

```
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: }
```

(c)



(d)

Fig. 6. 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 `Z` class that inherits from the `X` class. (c) The `Y` class that inherits from the `X` class. (d) The call graph for the functions in `X`, `Z`, and `Y`.

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.

3.4 Example of heuristic computation

We use the example in Fig. 6 to demonstrate how the heuristic values are computed. The class `X` in Fig. 6(a) is an abstract class with three methods: `aa`, `test`, and `bb`. The classes `Z` (Fig. 6(b)) and `Y` (Fig. 6(c)) inherit from the `X` class. In Fig. 6(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 Fig. 6(a); however, aa is a polymorphic method and the target of the call on line 8 of Fig. 6(a) depends on the type of x. The overall calling structure of the program is shown in Fig. 6(d). The test method in X can call the aa method in either the Z or Y class. The aa method then calls the cc method in its respective class. In the example shown in Fig. 6, the goal is to drive the program execution to line 9 in the cc method of the Y class in Fig. 6(c). In Fig. 6(c), if a certain condition is satisfied on line 8, a runtime exception is thrown on line 9.

Suppose for the program shown in Fig. 6, a main method calls test with different instances of X objects. During the static analysis phase, when we reach the test function in Fig. 6(a), the analysis accounts for the cost of moving through the *this*.bb method call on line 7 in Fig. 6(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 Fig. 6(a) and we want to compute a distance estimate to line 9 in Fig. 6(c). 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 Fig. 6 is x.aa(). We then call the get_estimate() function in Fig. 5 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 Fig. 6(d). Since the distance estimates in the aa method of the Y class have not been computed on the CFG (since the method does not 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.

In another example that demonstrates the dynamic computation of the heuristic, suppose the current location of the program is at line 4 in Fig. 6(c). 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 Fig. 4) on the aa method in Fig. 6(c). Note that since the call to this.cc() is a resolved polymorphic method call, the static analysis technique computes the cost of moving through the cc method at line 4 in Fig. 6(c). After refining the distance estimates, we return to the dynamic heuristic computation in Fig. 5. The analysis computes the distance estimates on the call trace

Y.aa() $\rightarrow$ Y.cc() based on the shortest distances in the CFGs of the methods `aa` and `cc` in the `Y` class.

4 Results

We conduct the experiments presented in this paper on a super-computing cluster of 618 nodes. Each node in the cluster has 8 GB of RAM and two Dual-core Intel Xeon EM64T processors (2.6 GHz). We run 100 trials of guided test, greedy best-first search, and randomized depth-first search to evaluate the effectiveness of the approach presented in this paper. All the trials are time bounded at one hour. This is consistent with other recent empirical studies [21, 23, 24]. We use the JavaPathfinder (JPF) model checker with partial order reduction turned on to conduct the experiments described in the paper.

Ranking Scheme: We use the two-level ranking scheme described in [17] to rank states in the guided test trials as well as in the greedy best-first search trials. In the greedy best-first trials, the primary key to sort the states in the priority queue is based on the number of locations observed in the sequence and the secondary key is based on the estimate generated by some other heuristic function. In the guided test, the states are ranked similarly but the search is guided in a greedy depth-first manner where the best ranked state among the immediate successors of a parent state is explored. When the test reaches a state with no successors, it backtracks to a certain point in the program and restarts the test. Note that the guided test too maintains a set of visited states.

In Table 1 we compare the guided test with greedy-best search and randomized depth-first search (DFS). For the three models with a given thread configuration in Table 1, we measure the error density, the number of states generated, the total time taken time, and memory used in the error discovering trials. The error density is defined as the ratio of error discovering trials to the total number of trials. The error density value of 1.00 indicates that the technique successfully finds the error in all the trials. The "time is seconds" is the total time taken (sum of time taken for static analysis and model checking). In the `Guided Test` we use the PFSM heuristic as the secondary heuristic function. To report results for the `Greedy Best-first`, we take pick the heuristic from the set of available heuristics with the best error discovery in the particular model using the two-level ranking scheme in a greedy-best first search. The set of heuristics we pick from are: prefer-thread heuristic, most-blocked heuristic, random heuristic, FSM heuristic, and the PFSM heuristic. For the `Reorder(8,1)` model the prefer-thread and PFSM heuristics have the best error discovery rate in the greedy best-first search. Both heuristics have comparable performance in error discovery in the `Reorder(8,1)` model. The random heuristic performs the best in the `AbsList(1,7)` model while the prefer-thread heuristic has the best performance in the `TwoStage(10,1)` model. The entries in the Table 1 with "-" indicate that the technique was not able to find the error in 100 trials, each time bounded at one hour. The `AbsList` example represents the race-condition

ERROR DENSITY

Subject	Guided Test	Greedy Best-first	Random DFS
Reorder(8,1)	1.00	0.54	1.00
AbsList(1,7)	1.00	0.66	0.01
TwoStage(10,1)	1.00	0.04	0.00

STATES

Subject	Guided Test			Greedy Best-first			Random DFS		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Reorder(8,1)	193	197	202	426	14579	232147	460500	2760116	4305479
AbsList(1,7)	727	727	727	42341	130035	322197	957981	957981	957981
Twostage(10,1)	329	335	340	413005	9874207	21965385	-	-	-

TIME IN SECONDS

Subject	Guided Test			Greedy Best-first			Random DFS		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Reorder(8,1)	0.34	0.39	0.54	0.48	10.75	166.88	191.52	1180.26	1966.55
AbsList(1,7)	4.77	4.81	5.08	31.98	83.06	169.62	706.30	706.30	706.30
Twostage(10,1)	0.43	0.46	0.52	96.31	1565.10	3526.55	-	-	-

MEMORY IN MB

Subject	Guided Test			Greedy Best-first			Random DFS		
	Min	Avg	Max	Min	Avg	Max	Min	Avg	Max
Reorder(8,1)	160	173	203	19	124	1526	358	415	444
AbsList(1,7)	170	191	248	877	1832	3069	153	153	153
Twostage(10,1)	163	181	203	1176	3934	5085	-	-	-

Table 1. Comparison of different search techniques.

found in the `AbstractList` class of the JDK 1.4 library and is described in the motivation section.

The results in Table 1 show that guided test is a better technique for error discovery compared to greedy best-first search (with an optimal secondary heuristic in a two-level ranking) and randomized DFS. In the three models, guided test has an error density of 1.00, while the error density values for the other search techniques vary between 0.00 to 1.00. A dramatic improvement in the guided test is observed in the states generated and total time taken. The guided test only generates 197 states, on average, in the `Reorder(8,1)` model while the greedy best-first search and random DFS generate 14,579 and 2,760,116 states respectively, on average, in error discovering trials. A similar trend is observed in the total time taken before error discovery among the guided test, greedy-best first search, and randomized DFS trials. The high memory usage in the guided search technique has been a deterrent in its application to finding errors in larger systems; however, the memory usage results in Table 1 show that the guided test uses comparable memory to randomized DFS with a better rate of error discovery in the given examples.

In Table 2 we specifically compare the performance of the PFSM heuristic with the FSM and random heuristic as a secondary heuristic in the two-level ranking scheme of guided test. These results are summary of the results presented in [17]. The random heuristic is the control to differentiate the performance gain obtained due to the sequence-based meta-heuristic versus the secondary heuristic to guide the test toward the next location. Note that we break all heuristic ties randomly [21]. The entries in Table 2 with "-" in the FSM heuristic columns indicate that the static analysis did not finish within the time bound of one hour. The FSM distance heuristic is a context-insensitive algorithm and performs an all-pairs shortest path analysis on the interprocedural CFG of the program to compute distance estimates. Note that the `AryList` model represents a race-condition in the `ArrayList` class of the JDK library. This error is manifested even while using synchronized `List` implementations.

The performance of guided test using the PFSM heuristic is dramatically better than the guided test using the random and FSM heuristic in a two-level ranking scheme. 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 static analysis cost.

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

5 Conclusions and Future Work

In this work we present a distance heuristic function that computes estimates in programs with unresolved polymorphic methods. The polymorphic distance heuristic uses the rapid type analysis to reduce the number of unresolved polymorphic method types in the program, performs an interprocedural static analysis to conservatively compute distances estimates, and dynamically refines the distance estimates after the types of polymorphic methods are resolved at transaction boundaries during model checking. The empirical analysis in this paper demonstrates that a guided test with the new heuristic performs better than a greedy-best first search with an optimal heuristic. The analysis also shows that the PFSM heuristic in a guided test setting outperforms the FSM distance heuristic that ignores the calling context information and the baseline random heuristic. In future we want 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 guided test. For example, to further refine 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

1. Fried, I.: Microsoft's Mundie looks beyond Gates (May 17, 2007) Interview on CNET News.
2. Sterling, N.: Warlock— a static data race analysis tool. In: USENIX Technical Conference Proceedings. (1993) 97–106
3. Flanagan, C., Leino, K.R.M., Lillibridge, M., Nelson, G., Saxe, J.B., Stata, R.: Extended static checking for Java. In: PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation, New York, NY, USA, ACM (2002) 234–245
4. 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
5. 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
6. Hovemeyer, D., Pugh, W.: Finding bugs is easy. SIGPLAN Not. **39**(12) (2004) 92–106
7. Artho, C., Biere, A.: Applying static analysis to large-scale, multi-threaded java programs. In: ASWEC '01: Proceedings of the 13th Australian Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2001) 68
8. Holzmann, G.J.: The SPIN Model Checker: Primer and Reference Manual. Addison-Wesley (2003)
9. Visser, W., Havelund, K., Brat, G., Park, S.: Model checking programs. In: 15th International Conference on Automated Software Engineering (ASE 2000), Grenoble, France (September 2000)

10. Ball, T., Rajamani, S.: The SLAM toolkit. In Berry, G., Comon, H., Finkel, A., eds.: 13th Annual Conference on Computer Aided Verification (CAV 2001). Volume 2102 of Lecture Notes in Computer Science., Paris, France, Springer-Verlag (July 2001) 260–264
11. Robby, Dwyer, M.B., Hatcliff, J.: Bogor: An extensible and highly-modular model checking framework. ACM SIGSOFT Software Engineering Notes **28**(5) (September 2003) 267–276
12. Sen, K., Agha, G.: CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In: Proc. 18th International Conference on Computer Aided Verification. (2006) 419–423 (Tool Paper).
13. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software verification with Blast. In Ball, T., Rajamani, S., eds.: Proceedings of the 10th International Workshop on Model Checking of Software (SPIN). Volume 2648 of Lecture Notes in Computer Science., Portland, OR (May 2003) 235–239
14. Valmari, A.: Stubborn sets for reduced state space generation. In: Proceedings of the 10th International Conference on Applications and Theory of Petri Nets, London, UK, Springer-Verlag (1991) 491–515
15. Flanagan, C., Godefroid, P.: Dynamic partial-order reduction for model checking software. In: POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, New York, NY, USA, ACM (2005) 110–121
16. Emerson, F.A., Sistla, A.P.: Symmetry and model checking. Formal Methods in System Design: An International Journal **9**(1/2) (August 1996) 105–131
17. Rungta, N., Mercer, E.G.: Guided test for detecting concurrency errors (2008) Available online at http://vv.cs.byu.edu/publications/papers/guided-test.pdf.
18. Edelkamp, S., Mehler, T.: Byte code distance heuristics and trail direction for model checking Java programs. In: Workshop on Model Checking and Artificial Intelligence (MoChArt). (2003) 69–76
19. Rungta, N., Mercer, E.G.: A context-sensitive structural heuristic for guided search model checking. In: 20th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, California, USA (November 2005) 410–413
20. Rungta, N., Mercer, E.G.: An improved distance heuristic function for directed software model checking. In: FMCAD '06: Proceedings of the Formal Methods in Computer Aided Design, Washington, DC, USA, IEEE Computer Society (2006) 60–67
21. 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
22. Bacon, D.F., Sweeney, P.F.: 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, New York, NY, USA, ACM (1996) 324–341
23. Rungta, N., Mercer, E.G.: Hardness for explicit state software model checking benchmarks. In: 5th IEEE International Conference on Software Engineering and Formal Methods, London, U.K (September 2007) 247–256
24. Dwyer, M.B., Person, S., Elbaum, S.: 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, New York, NY, USA, ACM Press (2006) 92–104