2006-11-01

An Improved Distance Heuristic Function for Directed Software Model Checking

Eric G. Mercer
eric_mercer@byu.edu

Neha Rungta

Original Publication Citation
N. Rungta and E. G. Mercer, "An Improved Distance Heuristic Function for Directed Software Model Checking", in Proceedings of Formal Methods in Computer Aided Design (FMCAD), San Jose, USA, pages 6-67, November 26.

An Improved Distance Heuristic Function for Directed Software Model Checking

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

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

***Abstract*— State exploration in directed software model checking is guided using a heuristic function to move states near errors to the front of the search queue. Distance heuristic functions rank states based on the number of transitions needed to move the current program state into an error location. Lack of calling context information causes the heuristic function to underestimate the true distance to the error; however, inlining functions at call sites in the control flow graph to capture calling context leads to an exponential growth in the computation. This paper presents a new algorithm that implicitly inlines functions at call sites to compute distance data with unbounded calling context that is polynomial in the number of nodes in the control flow graph. The new algorithm propagates distance data through call sites during a depth-first traversal of the program. We show in a series of benchmark examples that the new heuristic function with unbounded distance data is more efficient than the same heuristic function that inlines functions at their call sites up to a certain depth.**

I. Introduction

Multi-core processor design and hyper-threading create a need for techniques to validate concurrent interactions in threaded software artifacts. Traditional validation techniques based on test vector generation generally break down in the presence of concurrency since they cannot control scheduling decisions imposed by the operating system when running the input vectors. As a consequence, the validation is not effective in discovering subtle race or deadlock conditions that often lead to unexpected program behavior. Model checking is particularly effective in finding errors in deep execution traces because it considers all possible thread schedules in its analysis. Model checking has the potential to aid software validation if it can be effectively applied to real software artifacts.

State explosion is inherent in model checking, and it is especially problematic in software model checking because of the size and complexity of typical software artifacts. The process of model checking systematically explores the behavior space of the program in some way. There are several different tools and approaches to address the state explosion problem in software model checking, [1]–[5], and the work in this paper specifically focuses on directed model checking [6]–[8].

Directed model checking guides the search into areas of the state space where errors are more likely to exist. It aims to find a property violation before computation resources are exhausted. Directed model checking uses a heuristic function to rank successor states during state space exploration. The search order follows the ranking of states on the search frontier using a priority queue rather than a search stack. An *accurate heuristic function* reduces the number of states generated before error discovery without dramatically decreasing the frequency of state generation.

Early heuristics use notions from circuit design technology for computing the distance estimate. For example, hamming distance heuristics use the explicit state representation to estimate a bit-wise distance between the current state and an error state [9]. Current heuristic functions for directed software model checking are broadly classified into two categories: property based heuristics and structural heuristics. A property based heuristic function tries to estimate the number of changes in the program values needed to violate a property, while structural heuristics consider the structure of either the actual program or its resulting transition system to compute the heuristic estimate. Examples of property based heuristics are in [10], [11]. The work in this paper focuses on structural heuristics.

The notion of structural heuristics is introduced in [12]. The heuristics in [12] exploit the structural properties of thread interdependencies specific to only Java programs to find concurrency errors. Distance heuristic functions [13]–[15] are structural heuristics that compute the minimal number of transitions required to reach an error location from the current state in the control flow representation of the artifact. These heuristic functions have been shown to be effective in driving threads into race and deadlock conditions.

The extended-FSM (EFSM) distance heuristic combines statically computed distance estimates from the structure of the software artifact with the dynamic call trace in the run-time stack extracted from the state representation of the software artifact to improve the accuracy of the heuristic values [15]. The heuristic function is based on the following notion: at a given program location, the program is either going to reach a return point for the callee without encountering an error and return to the caller; or it encounters an error before it reaches the return point and does not return to the caller. The algorithm to compute the EFSM distance heuristic uses a graph with bounded calling context to compute the distances

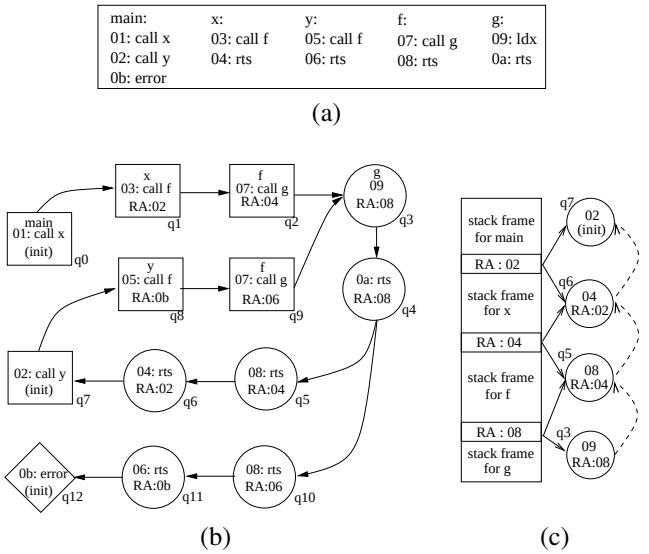


Fig. 1. A program and analysis demonstrating underestimation in the EFSM heuristic due to bounded calling context. (a) A program with a nested call depth of four. (b) An one-bounded CFG for the program that inlines procedures at call sites. (c) The run-time stack for a given state of the program.

in the forward direction. To build such a graph, procedures are inlined at call sites up to the depth of the user specified bound. Although the EFSM heuristic function reduces the number of states before error discovery in various examples, it is not an efficient heuristic function because the time required to construct the bounded graph increases exponentially with the bound; thus, the heuristic function does not scale well to programs with deeply nested function calls where large bounds are required for accurate heuristic estimates.

This paper presents a new *full context aware* (FCA) algorithm that implicitly inlines functions at call sites to compute distance data with unbounded calling context that is polynomial in the number of nodes in the *control flow graph* (CFG) for the software artifact. The new algorithm computes full context information for non-recursive programs with resolved function pointers, and works by propagating distance data through call sites during a depth-first traversal of the program's CFG. We show, in a series of benchmark examples, that a new heuristic function, e-FCA that is based on the EFSM heuristic but uses the FCA for forward distance estimates, generates fewer states and decreases total running time compared to the EFSM heuristic function that uses the inlined bounded calling context information [15].

II. Motivating Example

We demonstrate with an example how the accuracy of the EFSM heuristic in [15] relies on the bounded calling context used while computing the static shortest-path distances for the forward analysis. A program with a maximum possible call depth of four is presented in Fig. 1(a) where an error location, $0b$, is reachable from the `main` procedure only after making a call to procedures `x` and `y`. Procedures `x` and `y` both make calls to procedure `f` which in turn calls procedure `g`. In

```
procedure Extended_FSM(pc, rstack)
 1: d = 0, D_e = ∅
 2: while (rstack) do
 3:     ret_locs = get_entries(rstack, k)
 4:     n = get_node_in_k_bounded_CFG(pc, ret_locs)
 5:     E = {FSM(n, n_e) + d | n_e ∈ Errors ∧ in_scope(n_e, n)}
 6:     D_e = D_e ∪ E
 7:     n_end := return_statement(n)
 8:     d = d + FSM(n, n_end) + 1
 9:     pc = rstack.top()
10:     rstack.pop()
11: return min(D_e)
```

Fig. 2. Pseudo-code for the EFSM algorithm.

our program, the EFSM heuristic function tries to accurately estimate the minimal number of transitions required to reach the error location from the current program location. For example, from line 01 of `main`, it computes the number of instructions that need to be executed in order for the program state to reach the error location in line $0b$ of `main`. Ideally, the heuristic computation needs to account for the fact that the true execution flow of the program moves through procedures `x` and `y` before reaching the error.

The EFSM heuristic inlines procedures at call sites up to a bounded depth to capture partial context information. It does this by constructing a k-bounded CFG in a depth-first traversal of the program, where k is the specified bound. Each node in a k-bounded CFG is a location in the program with up to k entries for the partial call trace of length k used to arrive at that location. A one-bounded CFG for the program in Fig. 1(a) is shown in Fig. 1(b), where boxes represent call sites, circle nodes are return points or arbitrary program instructions, and the diamond shape nodes represent the error locations in the artifact. Each node, regardless of its type, has a program location identifier to map it back to the original program followed by a return address indicated by the `RA` label. There is a single return address in each node for this example because the k-bound of the graph is one. Returning to our example, procedures `x`, `y`, and `f` have enough context information to be uniquely inlined at their call sites; however, procedure `g` is not fully inlined at its call sites because unlike procedure f that is called from two unique call sites: locations 03 and 05, while procedure `g` is invoked two times from the same call site: location 07. In Fig. 1(b), the node q_4, an `rts` (return) instruction, is at program location $0a$ in procedure `g`. The return instruction can transfer control to any node that is at location 08 in procedure `f`. The edges from the q_4 node show that there are two possible return points: nodes q_5 and q_{10}. Both are at location 08, and both are an invocation of `f`. Without a k-bound of at least two, there is not enough context to create a unique invocation in the graph of the partial call trace, `f` $\rightarrow$ `g`, for both the `x` and `y` originations. The missing context leads to an underestimation of the final estimate in a shortest-path analysis because the shortest-path analysis uses the `x` invocation to get to procedure `f` but returns to the `y` invocation.

The EFSM heuristic function in [15] uses the calling context in the run-time stack present in the state of the program to recapture part of the missing calling context in the k-bounded CFG. There is no additional overhead in maintaining the run-time stack of the program because at any point in the program, the model checker has a complete snapshot of the actual state of the program including its entire run-time stack. The concrete run-time stack reflects the complete call trace from the top-most procedure of the artifact to the current program location which can be used in conjunction with the k-bounded graph to produce an accurate distance estimate. This is done by unrolling the run-time stack, and at each stack frame, considering the case that the program moves forward to encounter an error without returning from the current stack frame, or it returns from the current stack frame and then moves forward to encounter an error. The heuristic function minimizes over each of these scenarios as it moves through stack frames.

In Fig. 1(c), we present an example of a run-time stack from the concrete state of the program in the model checker. The current stack frame is for procedure `g` shown at the bottom of the run-time stack since it grows downward. The return address for the current procedure in the run-time stack is location 08 in procedure `f`. The current program location is 09 in the procedure `g`. The EFSM heuristic function, shown in Fig. 2, takes the current program location (pc) and combines it with the first k return locations (ret_locs) from the run-time stack ($rstack$) to identify the node (n) corresponding to the current program state in the k-bounded CFG (lines 3-4). The corresponding node in the one-bounded graph for the current program state of Fig. 1(b) is q_3 since it is at program location 09 with a return address of 08. The heuristic function in Fig. 2 now computes a distance estimate in the forward direction within the scope of the current stack frame (line 6). It requests a distance estimate on the k-bounded graph to all possible errors ($n_e \in Errors$) in the forward direction using a shortest-path analysis ($\mathsf{FSM}(n, n_e)$) assuming the current procedure does not return from the current stack frame ($\mathsf{in\_scope}(n_e, n) = true$). In our example, the error is not reachable from procedure `g` without moving through its return point so the shortest-path analysis returns ∞. The heuristic function in Fig. 2 then makes a note of this distance (line 6) and then computes the shortest distance to the previous call frame through the return statement of the current procedure (lines 7-8). It finally simulates returning from the current call frame by making the first return location its current program location (line 9).

Continuing with the concrete example in Fig. 1(c), after returning from procedure `g`, the EFSM heuristic function combines the new current program location, now 08, and the return location, 04, to find the next node in the k-bounded CFG. This corresponds to node q_5 in the one-bounded graph in Fig. 1(b). The heuristic function requests another forward estimate which is still ∞. It then considers the cost of returning from the stack frame. The algorithm in Fig. 2 repeats this process until it runs out of stack frames in the run-time stack (line 2), and it then reports the distance estimate to the nearest error computed during the analysis (line 11). The key observation in this example is that the call site for procedure `g` is resolved using the run-time stack, and the EFSM heuristic reports the correct distance value from node q_3 to the error location. This does not, however, completely remove the underestimation in other distance estimates.

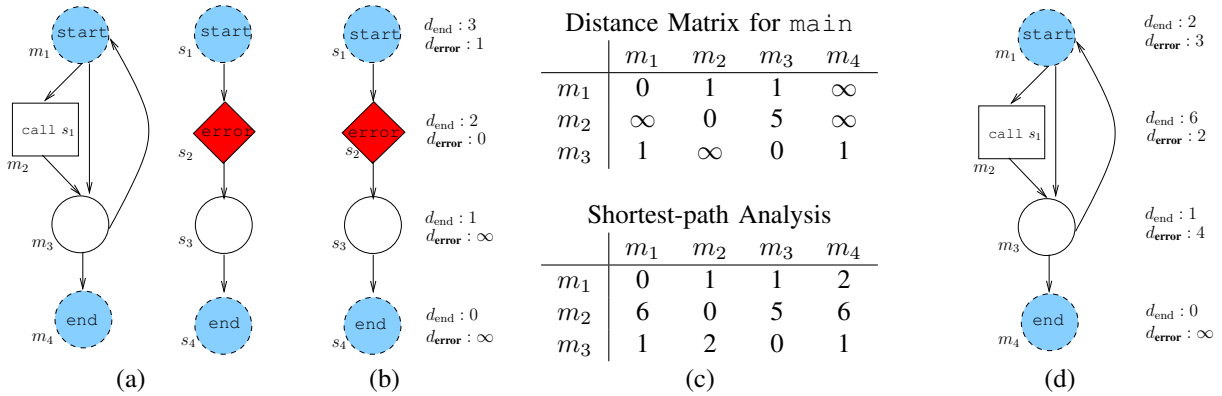
When the program execution is at the topmost level of the call structure or has a shallow call depth, the heuristic estimate computation is reduced to a shortest-path analysis on the k-bounded CFG. Returning to our example, suppose the concrete state in the model checker has a single frame in the run-time stack showing the current location to be node q_0. A request for the distance estimate in the forward direction returns a distance of seven which is the shortest-path to the error node q_{12} from node q_0 in the one-bounded CFG. The missing context information at depths greater than one is needed to resolve the unique call sites leading to node q_4, and it causes the shortest-path analysis to choose a path that is not consistent with the actual program execution from node q_0.

The cost of building the k-bounded graph is exponential in the nested call depth due to inlining. The cost of doing the shortest-path analysis is also very expensive since it is run several times to account for the scoping check on the path to the error. For the EFSM heuristic to be accurate, it needs full calling context, but for it to be efficient in runtime, it needs a small k-bound. The goal of this work is to produce an efficient estimate in terms of its accuracy and computational overhead.

III. Full Context Aware (FCA) Algorithm

The FCA algorithm is an interprocedural control flow analysis technique that implicitly constructs call traces to compute static lower-bounds on distance estimates to return locations of the procedures and nearest error locations in a software artifact using shortest-path analysis and depth-first traversal. The FCA algorithm uses the reverse invocation order to summarize the shortest-path analysis in all the callees of a given procedure. It then propagates the summarized distance information of the callees back to the given procedure. The input to the algorithm is a set of CFGs with a single CFG for each method or procedure in the artifact. Fig. 3(a) is an example of the input for a program with two procedures `main` and `sub1`. Each CFG has a single start node and end node. A call node is represented as a box in the CFG. The label in the call node identifies the start node for the target CFG of the call. The diamond shape nodes represent error nodes in the program. These are most often critical sections or assertion points in the software artifact. We associate values with each CFG node for the distance to the end node (d_{end}) and the distance to the nearest error node (d_{error}).

The FCA algorithm uses a depth-first traversal to build a distance matrix for a given CFG to use in a shortest-path analysis to compute d_{end} and d_{error} for each node. We associate with each individual CFG a distance matrix, L, that is defined over the number of nodes in the CFG. L is initialized with entries along the diagonal set to zero and all other entries



Distance Matrix for `main`

	m_1	m_2	m_3	m_4
m_1	0	1	1	∞
m_2	∞	0	5	∞
m_3	1	∞	0	1

Shortest-path Analysis

	m_1	m_2	m_3	m_4
m_1	0	1	1	2
m_2	6	0	5	6
m_3	1	2	0	1

(c)

Fig. 3. Execution of the FCA on a set of CFGs. (a) A set of two CFGs for a software artifact. (b) The CFG for `sub1` annotated with d_{end} and d_{error} data. (c) Distance matrix L for `main` before and after shortest-path analysis. (d) The CFG for `main` annotated with d_{end} and d_{error} data.

set to ∞. Edge costs in L are added as the depth-first traversal moves through nodes in the CFG. The depth-first traversal begins at the start node of the top-level method in the software artifact. In our example, the traversal starts at node m_1, updates the (m_1, m_2) entry in L to add the cost of the $m_1 \rightarrow m_2$ edge, and then visits m_2. The distance between two immediate successors in a CFG is one for all nodes except the successors of call nodes. The distance to the immediate successor of a call node needs to reflect the cost of moving through the target CFG of the call.

The depth-first traversal moves to the start node of the target CFG at a call node, and when the traversal returns, it then explores the immediate successor of the call node in the current CFG. The distance to the immediate successor of the call node relies on the analysis of the target CFG to account for the cost of moving through the target CFG without encountering an error. The distance to the immediate successor of the call node is the d_{end} value stored in the start node of its target CFG plus two: one to move to the start node of the target CFG and one to return from the end node of the target CFG. Fig. 3(b) shows the d_{end} and d_{error} values in the `sub1` CFG when the traversal returns to m_2 after analyzing `sub1`. Recall that call node m_2 points to the s_1 start node. The d_{end} value for s_1 is three. This reflects the number of program steps required in `sub1` to reach its s_4 end node. The traversal updates the (m_2, m_3) entry in L by setting the $m_2 \rightarrow m_3$ edge to five (three plus two), and continues the traversal by visiting node m_3. The top matrix in Fig. 3(c) is the final L matrix for `main`. The matrix includes all the edges in the `main` CFG with the cost of moving through `sub1` included in the row for m_2. Note that the matrix excludes the row for m_4 since the end node has no successors in the traversal. The L matrix provides the requisite data to derive d_{end} and d_{error} values for the CFG nodes.

The FCA algorithm computes d_{end} for each node in the current CFG with a shortest-path analysis using the distance matrix when the traversal is ready to backtrack out of the start node. Recall that L only includes nodes in the immediate CFG since call nodes move to the immediate successor in the CFG with the cost of moving through the target CFG. The bottom matrix in Fig. 3(c) is the final L matrix for `main` after the shortest-path analysis. Each d_{end} value for the CFG of `main` is set to its corresponding entry in the m_4 column of L after the shortest-path analysis. This is the shortest-path through the CFG to the end node m_4 including the cost of function calls.

The nearest error distance values are computed by minimizing over distances to error locations in the current CFG and distances to error locations reachable from the target CFGs of call nodes. In the discussion, N_{error} and N_{call} denote the sets of error locations and call nodes respectively for a given CFG. We use the results of the shortest-path analysis in the distance matrix, L, to compute distances to the nearest error location. For convenience, a global array, $D_{\text{error}}(n)$ is used to store the distance to the nearest error, d_{error}, for node n in a CFG. The equations for computing d_{error} for a given node n are

$$d_{\text{local}} = \min_{n_e \in N_{\text{error}}} (L(n, n_e)) \quad (1)$$

$$d_{\text{nonlocal}} = \min_{n_c \in N_{\text{call}}} (L(n, n_c) + D_{\text{error}}(n'_{\text{start}}) + 1) \quad (2)$$

$$d_{\text{error}} = \min(d_{\text{local}}, d_{\text{nonlocal}}) \quad (3)$$

$$D_{\text{error}}(n) = d_{\text{error}} \quad (4)$$

where n is the current node being analyzed and a call node n_c points to the start node n'_{start} of its target CFG.

The d_{local} value in Equation 1 is the distance to the nearest error in the immediate CFG. This error is reachable without having to move into a different CFG through a call node. The value is taken directly from the shortest-path analysis results in the distance matrix ($L(n, n_e)$ is the shortest-path from node n to node n_e in the current CFG).

The d_{nonlocal} value in Equation 2 is the distance to the nearest error through a call node of the immediate CFG. This error is only reachable by moving into a different CFG through a call node. The equation computes the transitive distance of first moving forward to the call node and then moving from the start node of the target CFG to the error. In the equation for the n_c call node, n'_{start} is the start node of its target CFG. The value stored in $D_{\text{error}}(n'_{\text{start}})$ is computed prior by virtue of the depth-first traversal. The traversal only triggers the distance

```
procedure compute_distances(N, E, n_start, n_end, N_error, N_call)
 1: /* Visited is global variable initialized to ∅ */
 2: if n_start ∉ Visited then
 3:     Visited = Visited ∪ {n_start}
 4:     /* L : N × N → ℕ ∪ {∞}, entries along the diagonal are set
        to 0, while all other entries are set to ∞ */
 5:     L = traverse_CFG(n_start, N_call, L)
 6:     L = compute_all_pairs_shortest_distance(L)
 7:     for each n ∈ N do
 8:         d_end = L(n, n_end)
 9:         /* D_end, a global array of size X is initialized to 0 */
10:         D_end(n) = d_end
11:         d_local = min_{n_e ∈ N_error}(L(n, n_e))
12:         ⟨N', E', n'_start, n'_end, N'_error, N'_call⟩ = Target(n_k)
13:         d_nonlocal = min_{n_k ∈ N_call}(L(n, n_k) + d_error(n'_start) + 1)
14:         d_error = min(d_local, d_nonlocal)
15:         /* D_error, a global array of size X is initialized to 0 */
16:         D_error(n) = d_error
17: return
18:
procedure traverse_CFG(n_x, N_call, L)
19: if n_x ∈ N_call then
20:     ⟨N', E', n'_start, n'_end, N'_error, N'_call⟩ = Target(n_x)
21:     compute_distances(N', E', n'_start, n'_end, N'_error, N'_call)
22:     d_succ = D_end(n'_start) + 2
23: else
24:     d_succ = 1
25: /* Conditional branches have multiple successors */
26: for each n'_x ∈ succ(n_x) do
27:     L(n_x, n'_x) = d_succ
28:     L = traverse_CFG(n'_x, N_call, L)
29: return L
```

Fig. 4. Pseudo-code for the FCA algorithm.

analysis when it is ready to backtrack out of a start node to resolve call dependencies in the computation. The final d_{error} value for node n in Equation 3 is either local to the CFG or reached through a call node in the CFG.

Fig. 3(d) shows the end results of the FCA algorithm for `main`. Fig. 3(b) and Fig. 3(d) give the complete view of the final analysis. For Fig. 3(d), the path to the end node for m_1 bypasses the m_2 call node for a distance of two, which is the number of edges in the path $m_1 \rightarrow m_3 \rightarrow m_4$. An error location is only reachable through the target CFG of the call node m_2. The nearest error for m_1 is three which represents the path $m_1 \rightarrow m_2 \rightarrow s_1 \rightarrow s_2$ in the CFGs. An error location cannot be reached from node m_4.

The FCA algorithm lower-bounds all distance estimates by assuming shortest-paths through CFGs. From this, the d_{error} and d_{end} data by themselves form an admissible and consistent distance estimate similar to the finite state machine (FSM) distance heuristic in [13]. Regardless of the true path of execution, the length of that path is at least that of the shortest-path through the CFG. An example is seen in Fig. 3(d) where the algorithm bypasses the m_2 call node to reach the m_4 end node. If the true path of execution follows m_2, then the actual distance is strictly larger than the reported distance. This lower-bound also appears in all iterative constructs of the CFG.

The pseudo-code for the FCA algorithm is presented in Fig. 4. In Fig. 4, a *CFG* is a tuple $\langle N, E, n_{\text{start}}, n_{\text{end}}, N_{\text{call}}, N_{\text{error}} \rangle$ where N is set of uniquely labeled nodes, $E \subseteq N \times N$ is the set of edges, $n_{\text{start}} \in N$ is a unique start node, $n_{\text{end}} \in N$ is a unique end node, $N_{\text{call}} \subseteq N$ is a set of call nodes, and $N_{\text{error}} \subseteq N$ is a set of error nodes. D_{end} and D_{error} are global arrays that store distances to the end node and nearest error node respectively for $X = |\bigcup_{1 \leq i \leq m} N_i|$ nodes where m is the number of procedures in the artifact. Note that the d_{error} and d_{end} values stored in D_{error} and D_{end} arrays are the same d_{error} and d_{end} values annotated on the CFGs. The function Target takes a call node as input and returns the target CFG of the call node. Finally, the function $\text{succ}(n_x) = \{n_y \in N \mid (n_x, n_y) \in E\}$, which means the succ function returns a set containing all the immediate successors of the input node, n_x.

In Fig. 4, the compute_distances function intializes a distance matrix L (line 4) for the input CFG and calls the traverse_CFG function (line 5). The traverse_CFG function uses a depth-first traversal to add edge costs between successors of the CFG in the distance matrix L (lines 26-28). If the traversal encounters a call node (line 19), the algorithm makes a mutually recursive call to compute_distances with the target CFG of the call node (line 21). When the execution returns, it adds the edge cost to the immediate successor of the call node taking into account the cost of moving through the target CFG without encountering an error (line 22). For all other nodes, the distance between two immediate successors in a CFG is one (line 24). After the traversal of the CFG is done, the algorithm returns the distance matrix L (line 29), and the flow of execution returns to the compute_distances function where the FCA algorithm computes the all-pairs shortest-path on the distance matrix L (line 6). For each node in the CFG, it adds the corresponding d_{end} and d_{error} values to the global arrays D_{end} and D_{error} (lines 7-16). The d_{error} value is computed by minimizing over distances to error locations in the current CFG (d_{local}) and distances to error locations reachable from the target CFGs of call nodes (d_{nonlocal}).

The intial traversal and the final algorithm to propagate the d_{end} and d_{error} values are linear in the number of nodes in the artifact since the traversal reuses the information from the secondary analysis if it encounters the same CFG numerous times from different call nodes in the artifact. The complexity of the secondary analysis is $O(X^3)$, where X is the total number of nodes in the artifact because an all-pairs shortest-path algorithm is run once for every reachable CFG in the artifact. Hence, the complexity of the FCA algorithm is polynomial in time and space with regards to the total number of nodes in the CFGs as a result of the shortest-path analysis.

IV. EFSM WITH FCA

The FCA is a forward analysis algorithm which is run once statically. The FCA cannot resolve the non-determinism arising from the end nodes in the CFGs of the program. For example, in Fig. 1(a) there are two calls to procedure `f`. The CFG of `f` does not contain any information about where the flow of execution returns when it exits procedure `f`. The only information present in the nodes of the CFGs are the distance values to the end (d_{end}) of the CFGs and the distance values to error locations (d_{error}) in the forward direction without

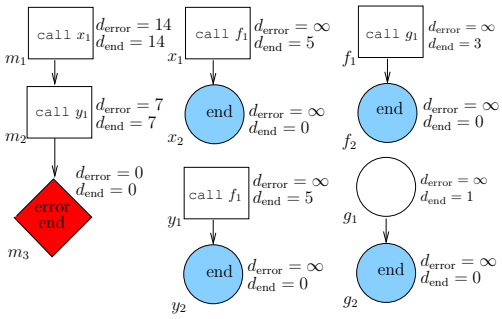


Fig. 5. The CFGs for the program in $Fig.\ 1(a)$ annotated with context aware distances after running the FCA algorithm.

executing the end of the CFGs. We dynamically recreate the call trace based on the values of the run-time stack like the EFSM heuristic to compute true successors of end nodes for a particular program execution path.

For the program shown in Fig. 1(a), the CFGs for each procedure annotated with d_{error} and d_{end} values after executing the FCA computation are shown in Fig. 5. Now let us consider a concrete example of how the EFSM is combined with the FCA to compute accurate heuristic estimates. Suppose the values on the return stack are: $\langle x_2, m_2 \rangle$ where the x_2 is the first return location encountered on exiting the current stack frame, and m_2 is the next return location. The current location of the program is f_1 in procedure `f`. The heuristic function in the EFSM requests a distance estimate to the error in the forward direction without exiting from the current procedure. Instead of computing this estimate on a k-bounded graph, it now considers the d_{error} value present on node f_1. In Fig. 5, we can see that the value of d_{error} is ∞ for node f_1 which means the error is not reachable in the forward direction.

After noting the value of d_{error} at node f_1, the heuristic function uses the value of d_{end} to compute the distance to the end node of procedure `f`. It requires this value to estimate the distance to the previous call frame. The algorithm to compute the heuristic estimate has an accumulator variable, $path$, that keeps track of the cost incurred in backtracking through the call frames. In Fig. 5 the value of d_{end} is three; the heuristic function adds one to it to account for the return and sets $path$ equal to four. Next, the heuristic simulates returning from the current call frame by making the first value on the return stack, x_2, its current program location like the EFSM algorithm. The value of d_{error} on node x_2 is ∞ showing that the error is still not reachable. Since x_2 is an end node, the value of d_{end} is zero, and the heuristic function increments $path$ by one for the return, changing the value of $path$ to five. The heuristic function repeats the process of unrolling the stack by moving to node m_2, where the value of d_{error} is seven. It adds this value to $path$ to get the final value of twelve as the estimate of the distance to the error. At all points of forward computation to find the error location, and while going to the end node, the heuristic function has access to unbounded context information from the FCA algorithm resulting in a better lower-bound on the true estimate of the distance to the error which is also admissible and consistent.

TABLE I
TIME TAKEN IN SECONDS FOR STATIC ANALYSIS

Name	T	M	k	FSM	EFSM	FCA
Hyman	2	3	1	0	3	0
Hyman	2	4	1	1	11	0
Hyman	2	5	1	1	27	0
D-phil	2	2	1	1	76	0
D-phil C	2	2	1	0	76	0
D-phil	2	3	1	1	146	0
D-phil C	2	3	1	0	147	0
D-phil	2	4	0	1	4	1
D-phil C	2	4	0	1	4	1
D-phil	2	5	0	2	7	1
D-phil C	2	5	0	2	7	1
Barbers	3	3	1	1	41	0
Barbers	3	4	0	1	3	1
Barbers	3	5	0	1	4	1

To calculate the heuristic estimate for a concurrent program with multiple threads, the approach presented in [13] computes the distance to the nearest error location for each thread and sums up the individual estimates to create a final heuristic value. To ensure underestimation of the distance to the error, we modify the number of individual estimates summed together based on the property being verified. For example, the heuristic estimate for a $mutex$ violation is the sum of distances in two threads which have the shortest paths to the critical section compared to all the other threads. Now, consider the property which is a check to see if any thread reaches a certain location in the program, like an $assert$ statement. In such a case, the heuristic estimate is the smallest distance in the set of estimated distances from the current location to the desired location for each thread. Another useful property checked in concurrent programs is whether two or more threads are $deadlocked$. In this case, we take the summation of the distances from the current state to the error state for two or more threads that can lead to a $deadlock$ state. From this point forward in the presentation we refer to the combination of the FCA and EFSM approaches as the e-FCA heuristic function.

V. RESULTS

We implemented the e-FCA heuristic function in the gnu-debugger based model checker Estes, [8], and executed it on a benchmark set consisting of programs with concurrency errors. The results show that the e-FCA heuristic reduces the total number of states generated and also decreases the total running time before error discovery compared to the FSM and EFSM distance heuristics.

We focus specifically on three classical concurrency problems in our benchmark suite: `Dining Philosophers`, `Barbershop`, and `Hyman's mutual exclusion principle`. The results presented are from a Pentium III 1.5 Ghz processor with 2 GB of RAM and are run on Estes, with a 6.1.1 version of the gnu debugger, using the m68hc11 backend simulator. We report the wall clock time for the time

TABLE II

A COMPARISON ACROSS DIFFERENT SEARCH TECHNIQUES

Name	T	M	k	Total States Generated					Time taken in Seconds				
				DFS	Rand	FSM	EFSM	e-FCA	DFS	Rand	FSM	EFSM	e-FCA
Hyman	2	3	1	6,478	15,800	10,227	7,160	3,817	3	9	4	6	1
Hyman	2	4	1	16,190	59,796	41,791	21,909	13,529	7	28	17	21	5
Hyman	2	5	1	40,471	91,947	123,743	59,951	38,745	17	42	49	56	16
D-phil	2	2	1	157,436	475,184	53,897	4,594	1,626	71	318	31	79	1
D-phil C	2	2	1	19,769	11,497	18,148	1036	415	12	10	14	77	0
D-phil	2	3	1	$*$	452,092	54,725	13,830	3,816	$*$	292	28	155	3
D-phil C	2	3	1	157,818	95,009	8,575	3,348	1,015	102	75	5	149	1
D-phil	2	4	0	$*$	999,480	186,419	36,467	13,696	$*$	730	113	27	8
D-phil C	2	4	0	548,127	173,494	42,107	10,224	3,655	299	159	31	12	3
D-phil	2	5	0	$*$	$*$	334,198	400,474	55,876	$*$	$*$	178	388	32
D-phil C	2	5	0	$*$	370,656	861,319	142,350	14,755	$*$	294	680	136	11
Barbers	3	3	1	$*$	442,285	82,333	21,465	3,298	$*$	282	41	14	2
Barbers	3	4	0	$*$	939,828	73,940	75,635	13,118	$*$	576	38	47	7
Barbers	3	5	0	$*$	$*$	378,632	388,161	66,608	$*$	$*$	237	252	38

taken to do the static analysis and for the total running time of the program before error discovery, as well as the total states generated before error discovery.

For the benchmarks, we add procedures containing nested function calls that do not affect the property being verified to the base implementation of the concurrent programs. We then randomly insert calls to these procedures throughout the programs to derive examples with varying call structures and call depths in order to test the accuracy and efficiency of the e-FCA in computing context aware distances. To measure the accuracy of the e-FCA heuristic function we compare the e-FCA to the shortest-path analysis (FSM) and the EFSM distance heuristic. We also compare it with random search and an exhaustive depth-first search (DFS). We use best-first search rather than A^* to decrease the number of states expanded before error discovery; although best-first does not guarantee a shortest error trace like the A^* search. In our experiments, the length of the error traces generated by the best-first search are comparable to that of A^*. The results of the analyses are shown in Table I, Table II and Table III. In Table I and Table II the first column (`Name`), shows the concurrent program being verified; if the program is model checked at the C-level, where a single C instruction is considered atomic, a letter C is appended to the end of the program name. Otherwise, the program is model checked at the assembly-level where a single assembly-level instruction is considered atomic. The next column (`T`) indicates the number of threads in the program, the column (`M`) shows the maximum possible call depth of the program, and the column (`k`) is the value of the bound picked for the EFSM heuristic.

The static analysis time reported in Table I is the time taken in seconds during the period after the execution of the program starts and before the model checking run begins. The FCA analysis takes negligible amount of time. The average time taken by the FCA to complete static analysis (0.65 secs) is less than the average time (1 secs) taken by the FSM, even though the FSM does not consider calling context of the program at all. As we increase the k-bound for the EFSM distance heuristic, the cost to construct the inlined graph and do a shortest path analysis for checking whether the error is in scope grows exponentially. The high overhead of static analysis with larger k-bounds forces us to pick a bound of either one or zero for the EFSM distance heuristic in order to finish static analysis within a few minutes. In spite of picking low bounds of k for the EFSM computation, the time taken by EFSM to complete static analysis is significantly higher compared to the FCA. Note that even with a bound of zero, the EFSM dynamically recreates the call trace of the program; hence, it has more context than the FSM distance heuristic.

For each model, we report the total number of states enumerated before finding the error state and total running time to find the error in Table II. The '$*$' symbol in Table II shows that after generating a million states, the error state was still not found, and at that point, the search was terminated. For the random search, the heuristic value is set to a random value and the numbers reported for the total number of states and total running time numbers are averaged over 10 model checking runs.

The e-FCA gains a significant reduction in total states generated compared to the other heuristics and search techniques as shown on the left side of Table II. In Table II, we can see that DFS, an exhaustive search is mostly ineffective in finding the error. In seven out of fourteen examples it is unable to find the error within a million states; however, sometimes DFS happens to find the error quickly by chance as seen in the `Hyman` examples. In some cases, the EFSM generates more states than the FSM distance heuristic before finding the error. Our experiments with different k-bounds show that for some programs, the improvement in error discovery by the EFSM heuristic with increasing context is not always monotonic. For such programs, the EFSM heuristic does not perform well until the context information reaches a certain threshold.

The e-FCA also obtains a significant decrease in the total running time compared to the other search techniques as shown on the right side of Table II. The state reduction achieved by the EFSM is not enough to compensate for the

TABLE III
SCALABILITY ACROSS DIFFERENT THREADS

	Depth = 2		Depth = 5		Depth = 9	
T	States	Time	States	Time	States	Time
5	814	1	7,064	5	92,434	71
9	1,070	1	7,320	6	93,230	82
11	1,196	1	7,446	11	93,356	144
15	1,448	1	7,698	12	93,608	158
18	1,641	1	7,891	12	93,801	165
20	1,767	2	8,071	13	93,927	173
25	2,086	3	8,336	15	94,246	187
30	2,401	3	8,970	18	94,561	204
40	3,040	4	9,603	22	92,500	233
51	3,736	8	10,306	30	95,896	311

high cost of static analysis causing its total running time to increase dramatically compared to the e-FCA. While computing the heuristic estimate, the EFSM faces an additional overhead cost of extracting the call trace in the run-time stack from the start of the program to its current point. The e-FCA faces the same overhead of extracting the run-time stack; however, this cost is very effectively mitigated by the significant reduction in the states generated and low cost of static analysis resulting in a substantial decrease in the total running time of the e-FCA before error discovery.

We test the scalability of the e-FCA heuristic function by instrumenting our implementation of the `barbershop` problem to allow a variable number of threads (between 5 and 51). Additionally, we implement three versions of the problem with varying maximum possible call depths of two, five, and nine. The total number of states and total running time before error discovery for these examples are presented in Table III. The first column (`T`) in Table III indicates the number of threads created for the particular example. From the barbershop example, it seems that the e-FCA scales to multiple threads with a high degree of nested function calls. DFS and random search do not find the error in a million states for even the smallest model. The FSM distance heuristic and EFSM distance heuristic, with small k-bounds, do not find the error in a million states for most of the models. With a slightly higher k-bound the EFSM heuristic does not finish static analysis in 1 hour for any of the examples.

VI. CONCLUSION AND FUTURE WORK

In this paper we present the FCA algorithm that computes full context aware distances for a CFG in a non-recursive program with resolved function pointers by implicitly inlining function calls. It propagates context information through call nodes, start nodes, and end nodes of the CFG and annotates the nodes in the CFG with context sensitive distances to end nodes and error locations in the forward direction. We then present a new heuristic function, e-FCA which combines the unbounded distance data computed by the FCA algorithm with the dynamic recreation of the run-time stack from the EFSM heuristic function. The e-FCA heuristic function computes more accurate heuristic estimates compared to other distance heuristic functions.

In some cases, the e-FCA heuristic function underestimates the true distance to the error locations because it does not consider the feasibility of the execution paths. Resolving the feasiblity of all execution paths is not possible statically; however, while model checking, as the variables are assigned dynamic values, we can determine the infeasible execution paths. In future work we plan on pruning these infeasible execution paths before computing the heuristic estimate to overcome the underestimation arising due to the path-insensitive computation of the e-FCA.

REFERENCES

[1] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with Blast," in *Proceedings of the 10th International Workshop on Model Checking of Software (SPIN)*, ser. Lecture Notes in Computer Science, T. Ball and S. Rajamani, Eds., vol. 2648, Portland, OR, May 2003, pp. 235–239.

[2] T. Ball and S. Rajamani, "The SLAM toolkit," in *13th Annual Conference on Computer Aided Verification (CAV 2001)*, ser. Lecture Notes in Computer Science, G. Berry, H. Comon, and A. Finkel, Eds., vol. 2102. Paris, France: Springer-Verlag, July 2001, pp. 260–264.

[3] J. Penix, W. Visser, C. Pasaranu, E. Engstrom, A. Larson, and N. Weininger, "Verifying time partitioning in the DEOS scheduling kernel," in *22nd International Conference on Software Engineering (ICSE00)*. Limerick, Ireland: ACM, June 2000, pp. 488–497.

[4] Robby, M. B. Dwyer, and J. Hatcliff, "Bogor: An extensible and highly-modular model checking framework," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 267–276, September 2003.

[5] T. Ball and S. K. Rajamani, "Bebop: A symbolic model checker for boolean programs," in *7th International SPIN Workshop*, ser. Lecture Notes in Computer Science, K. Havelund, J. Penix, and W. Visser, Eds., vol. 1885. Springer, August 2000, pp. 113–130. [Online]. Available: citeseer.nj.nec.com/ball00bebop.html

[6] W. Visser, K. Havelund, G. P. Brat, S. Park, and F. Lerda, "Model checking programs." *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[7] P. Leven, T. Mehler, and S. Edelkamp, "Directed error detection in C++ with the assembly-level model checker StEAM," in *Proceedings of 11th International SPIN Workshop, Barcelona, Spain*, ser. Lecture Notes in Computer Science, vol. 2989. Springer, 2004, pp. 39–56.

[8] E. G. Mercer and M. Jones, "Model checking machine code with the GNU debugger," in *12th International SPIN Workshop*, ser. Lecture Notes in Computer Science, vol. 3639. San Francisco, USA: Springer, August 2005, pp. 251–265.

[9] C. H. Yang and D. L. Dill, "Validation with guided search of the state space," in *35th Design Automation Conference (DAC98)*, 1998, pp. 599–604. [Online]. Available: http://citeseer.nj.nec.com/yang98validation.html

[10] S. Edelkamp, A. L. Lafuente, and S. Leue, "Directed explicit model checking with HSF-SPIN," in *Proceedings of the 7th International SPIN Workshop*, ser. Lecture Notes in Computer Science, no. 2057. Springer-Verlag, 2001.

[11] K. Seppi, M. Jones, and P. Lamborn, "Guided model checking with a bayesian meta-heuristic." *Fundamenta Informaticae*, vol. 70, no. 1-2, pp. 111–126, 2006.

[12] A. Groce and W. Visser, "Model checking Java programs using structural heuristics," in *2002 ACM SIGSOFT International symposium on software testing and analysis*, 2002, pp. 12–21.

[13] 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)*, 2003, pp. 69–76.

[14] 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*, 2001, pp. 37–46. [Online]. Available: citeseer.ist.psu.edu/cobleigh01right.html

[15] 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*, Long Beach, California, USA, November 2005, pp. 410–413.