



Faculty Publications

2005-11-01

A Context-sensitive Structural Heuristic for Guided Search Model Checking

Eric G. Mercer
eric_mercer@byu.edu

Neha Rungta

Follow this and additional works at: <https://scholarsarchive.byu.edu/facpub>



Part of the [Computer Sciences Commons](#)

Original Publication Citation

N. Rungta and E. G. Mercer, "A Context-sensitive Structural Heuristic for Guided Search Model Checking", in Proceedings of 2th IEEE/ACM International Conference on Automated Software Engineering, Long Beach, USA, pages 41-413, November 25.

BYU ScholarsArchive Citation

Mercer, Eric G. and Rungta, Neha, "A Context-sensitive Structural Heuristic for Guided Search Model Checking" (2005). *Faculty Publications*. 344.
<https://scholarsarchive.byu.edu/facpub/344>

This Peer-Reviewed Article is brought to you for free and open access by BYU ScholarsArchive. It has been accepted for inclusion in Faculty Publications by an authorized administrator of BYU ScholarsArchive. For more information, please contact ellen_amatangelo@byu.edu.

A Context-sensitive Structural Heuristic for Guided Search Model Checking

Neha Rungta
Department of Computer Science
Brigham Young University
Provo, Utah 84602
neha@byu.edu

Eric G Mercer
Department of Computer Science
Brigham Young University
Provo, Utah 84602
egm@cs.byu.edu

ABSTRACT

Software verification using model checking often translates programs into corresponding transition systems that model the program behavior. As software systems continue to grow in complexity and size, exhaustively checking a property on a transition graph becomes difficult. The goal of guided search heuristics in model checking is to find a counterexample to the property being verified as quickly as possible in the transition graph. The FSM distance heuristic builds an interprocedural control flow graph of the program to estimate distance to a possible error state. It ignores calling context and underestimates the true distance to the error.

In this paper we build on the FSM distance heuristic for guided model checking by using the runtime stack to reconstruct calling context in procedural calls. We first build a more accurate static representation of the program by including a bounded level of calling context. We then use the calling context in the runtime stack with the more accurate control flow graph to estimate the distance to the possible error state. The heuristic is computed using both the dynamic and static construction of the program. We evaluate the new heuristic on models with concurrency errors. Experimental results show that for programs with function calls, the new heuristic better guides the search toward the error while the traditional FSM distance heuristic degenerates into a random search.

Categories and Subject Descriptors

D.2.4 [Software Engineering]: Software/Program Verification—*Model checking*

Keywords

Verification, validation, guided search, model checking, structural heuristics

1. INTRODUCTION

Formal verification is an important component in the validation process for capitol and safety critical systems. For example, embedded devices are often capitol critical systems because they are initially deployed in large quantities. The growing complexity of software for embedded devices challenges traditional strategies based on code coverage and vector simulation. Model checking verifies properties of the software in its dynamic environment to help prevent the escape of bugs that follow from an overlooked test scenario. In other words, model checking can reveal subtle violations of functional properties that would be difficult to reproduce *a priori* in vector based testing.

Model checking, intuitively, is a method to systematically explore behavior in a concurrent system to see if it meets user specified properties. It is the process of showing that a state transition graph is a model of a formula describing a correctness property. Formally, if M is a state transition graph, s is a state in M , and f is a correctness property, then $M, s \models f$ denotes that starting at state s , the state transition graph M is a model of f ; or in other words, the property f holds in M given s as a starting state. Model checking is the process by which the relation between M , s , and f is validated by proof that $M, s \models f$ or counterexample that $M, s \not\models f$.

Several techniques for software model checking are actively being pursued. One approach applies conservative abstractions to the high-level programming language to reduce the size and complexity of the state transition graph [18, 2]. This approach is particularly successful in verifying control intensive code in which conditional expressions do not depend on extensively manipulated data values. Another approach applies bounded model checking to C programs and can verify buffer overflows, pointer safety, user defined assertions, and language consistencies [6, 7]. Bounded model checking uses SAT methods to show the relation between M , s , and f in the model checking problem. Other approaches translate the software under test into the formally defined input language of an existing model checker [26, 27, 3]. Language extensions are sometimes needed to facilitate the translation since the language semantics are not always directly supported by the existing framework [19]. The Bandera project for Java uses Bogor that natively supports threads and memory management in its input language [9, 27]. A recent approach uses symbolic execution to verify properties of algorithms [25]. Counter example guided

abstraction refinement is very effective in managing state explosion and scales to large systems where data manipulation does not significantly affect control flow. If the system is largely data flow driven, however, abstraction refinement begins to break down.

Explicit state model checking directly explores the state transition graph M . The model checking proof of a property f traverses M from the starting state s looking for a violation. Several approaches exist for explicit state model checking of software. Java Pathfinder model checks the actual software using a Java virtual machine [30]. Similar approaches use simulators and debuggers for other machine architectures [20, 21]. These approaches retain a high-fidelity model of the target execution platform with low-level control of scheduling decisions. Other approaches work directly with the machine-code of the program to run at-speed analysis on the native hardware. Valgrind instruments the actual machine code [22, 24], and Verisoft runs the code in a scheduling environment as a process it manages [13]. Testing at speed can boost performance in state generation, and in the case of code instrumentation, the overhead has been shown to be acceptable in large programs [29, 23].

The primary challenge to explicit model checking is managing the size of the transition system. For large software systems, the computation resources are quickly exhausted before model checking finishes exploration. One solution to state explosion is through guided model checking. Guided model checking focuses on error discovery by using heuristics to prioritize the search of the transition system. The idea is to discover an error before computation resources are exhausted. Search priority is determined by heuristics that rank states in order of interest, with states estimated to be near errors being explored first. Hamming distance heuristics use the explicit state representation to estimate a bit-wise distance between the current state and an error state [32]. Hamming distance heuristics ignore the structure of the property being verified as well as the structure of the system. The structure of the property is accounted for in [11]. The approach is further refined with Bayesian reasoning in [28]. These heuristics only consider the state representation and structure of the property being verified.

Guided search heuristics can be improved by considering program structure with the property being verified. Heuristics related to Java programs are described in [15]. These heuristics are particularly interesting because they start to look more at the actual structure of the Java program in addition to the property being verified. Program structure gives insight to how values in the state can be affected relative to any given property being verified. Heuristics can refine estimates using knowledge of how and when the program manipulates data of interest.

Program structure is central to the finite state machine (FSM) distance heuristics in [12, 8]. FSM distance heuristics extract from the actual program a set of communicating finite state machines that represent the control flow of the program. The distance estimate is based on the current position of the program in its finite state machine representation and a location of where an error is checked for by the program in the same representation. The heuristic estimate

is the length of the shortest path between these two points in the FSM representation.

The FSM distance heuristic is not context sensitive; this means that it ignores the calling context of functions. Ignoring the calling context in function calls causes the heuristic to underestimate the actual distance to the error state. In the worst case, a guided search using the FSM distance heuristic degenerates into a random search of the transition system. Adding context sensitivity to the FSM distance heuristic, as done in this paper, reduces the number of states explored before error discovery in guided search model checking over its context free counterpart. In designs where the FSM distance heuristic is effective, we expect our new algorithm that considers calling context to outperform the original context-free heuristic in terms of states explored before error discovery.

In this paper, we present an algorithm that reconstructs calling context using the runtime stack in the concrete state with an augmented control flow graph to estimate FSM distance to an error state. Intuitively, the new algorithm uses the runtime stack to determine return points from function calls. In programs where the same function is invoked from several different call points, if call frames exist in the runtime stack for the invocations, then the heuristic correctly computes the return point from the call frames in computing the FSM distance heuristic. It does this until it runs out of call frames or arrives at the scope of the error location. If it runs out of call frames, it uses an augmented interprocedural control flow graph to estimate the remaining distance between the current program location and an error location. The new heuristic more accurately estimates the distance between the current program point and a point in the program where an error can be detected by using the call context to remove the false paths in the FSM representation. The better estimation improves error discovery. This is shown with a series of benchmark examples where the new heuristic visits fewer states before finding an error.

2. FSM DISTANCE HEURISTIC

The FSM distance heuristic builds a static representation of the program that depicts its flow of execution and structure. There are many ways to abstractly represent program flow and structure. Edelkamp and Mehler in [12] use a partitioning function to map object code into blocks. They use a target function to generate connecting edges between the blocks. This graph is identical in structure and function to an interprocedural control flow graph (ICFG). Hence we will refer to the graph in [12] as an ICFG. An ICFG has a control flow graph (CFG) for each of its procedures. These CFGs are connected at call sites and return points. Similar to the definition of the partition function in [12], we use the term *basic block* loosely to be either a maximal or non maximal sequence of instructions that can be entered only at the first instruction and exited only from the last instruction. For all of the examples in this paper, we treat each single instruction as a basic block for simplicity.

Definition 1. A *Control flow graph* (CFG) is a directed graph $G = (V, E)$. Vertices (V) are the basic blocks in a given procedure. The edges (E) represent possible flow of

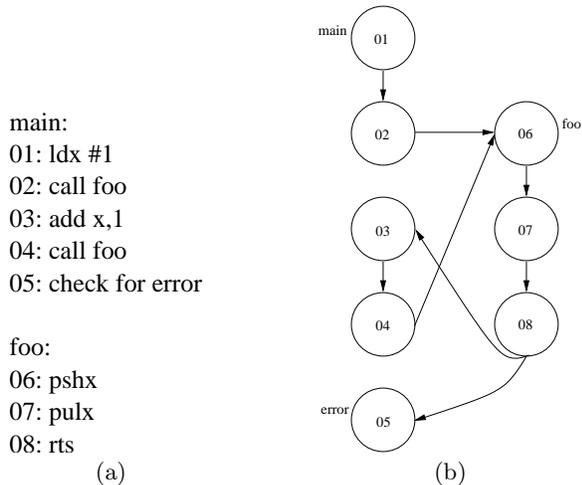


Figure 1: A simple example (a) A program that calls foo twice and checks for error (b) The control flow graph indexed on PC value

execution between the vertices. Each CFG has *begin* and *end* vertices. All the vertices in the CFG are reachable from *begin* and have a path to *end* [1].

Definition 2. We define an *Interprocedural Control Flow Graph (ICFG)* to be similar to the ones defined in [5, 17]. There are four special vertices defined in an ICFG: *entry*, *exit*, *call*, and *return*. The *entry* and *exit* vertices correspond to the *begin* and *end* vertices in the individual CFGs for each procedure in the system. The *call* vertices represents the call sites where procedures are invoked, and *return* vertices are the return points where execution resumes after a procedure completes. As we traverse the individual CFGs for each procedure, an edge is added from *call* vertices to the corresponding *entry* vertices of the CFGs for the named procedures in the call sites. Similarly, edges are added from the *exit* vertices in the CFGs back to the *return* vertices created by the call sites. Note that edges are created from a given CFG’s *exit* vertex to every *return* vertex created by a call to that CFG. There is no information in the ICFG to distinguish which call site invoked the procedure, while inside the procedure.

The assembly instructions of a simple C program with two function calls from `main` to `foo` is shown in Figure 1(a). Figure 1(b) is the ICFG for the program. For convenience, we label the nodes in the ICFG with the program counter (PC) values from the corresponding locations indicated to the left of the program. We assume these PC values to be unique labels for each line of the source program. The vertices labeled with 02 and 04 refer to instructions in the CFG of `main` that call the procedure `foo`. Similarly, the vertices labeled by 03 and 05 are the return points in the CFG for `main` created by the call sites. Vertices 01 and 06 are entry vertices to the procedures, and vertex 08 is the exit vertex for procedure `foo`. Only a portion of the procedure `main` is shown in Figure 1(a) for brevity. The procedure `foo` is called twice from procedure `main`.

The FSM distance is defined as the minimal number of operations required to reach a goal state from the current state [12]. The FSM distance heuristic maps the current state of the program onto a vertex in the ICFG during the guided search. It calculates the length of the shortest path to the error state from the current vertex in the ICFG. It returns this length as the heuristic estimate to guide the search [12]. For example, if the current state of the program is at PC value 06 in the search, then the corresponding location in the ICFG is at vertex 06. The minimum number of steps required to reach the error location at vertex 05 from vertex 06 is the FSM distance. In this example, the shortest path from vertex 06 to 05 in the ICFG is along the path $06 \rightarrow 07 \rightarrow 08 \rightarrow 05$, and it has a length of 4. This is the reported value of the FSM distance heuristic from the current state.

The return address on a runtime stack, which is present in the current state of the search, indicates the true call site for a procedure. The FSM distance heuristic as defined in [12] does not consider this in computing the length of the shortest path to the error state. This is part of the calling context that the heuristic ignores in its computation. For our example in Figure 1, when the search is in the function `foo`, the return address on the runtime stack can be either 03 or 05 for each of the call sites in `main`. If the actual call site is from line 02 in the program, then the return address is 03; thus, the FSM heuristic computes the shortest path between the current state and the error using a false path and underestimates the true length. For this example, consider the calling context from line 02, the true distance is computed along the path $06 \rightarrow 07 \rightarrow 08 \rightarrow 03 \rightarrow 04 \rightarrow 06 \rightarrow 07 \rightarrow 08 \rightarrow 05$. This gives an estimate of 8 transitions needed to reach an error state.

Another problem in the FSM distance heuristic relates to indirect jumps and indirect procedure calls. Whenever there is a block containing an indirect jump or procedure call, the ICFG contains an edge from the block to every other block that it might reach whenever the target address cannot be statically resolved. This creates a multitude of edges that introduce false paths to error states with very short lengths. To mitigate the impact of these edges, we assume indirect jumps only target entries in defined jump tables, and we assume indirect procedure calls only target valid entry points in procedures. As mentioned earlier, any targets that can be statically resolved are statically resolved. Finally, we do not consider exceptional control flow mechanisms such as interrupts in this work aside from normal scheduling operations for thread and process management. This means we do not consider in the ICFG the possibility of getting to the error state through an interrupt routine since interrupts can occur at any point in the program and return to any point in the program. Statically we cannot resolve where interrupts take place or return flow of execution; this induces us to take the most conservative approach and assume the above. This introduces many false shortest paths to the error state and degrades the performance of the heuristic.

The FSM heuristic [12] is admissible and consistent. A heuristic function $h(v)$ is said to be admissible on (G, Γ) iff $h(v) \leq h'(v)$ for every $v \in G$. G is a directed graph, and Γ is the set of possible error states; while h is the estimated cost,

and h' is the true cost. A heuristic function $h(v)$ is said to be consistent (or monotone) on G iff, for any pair of vertices v' and v , the triangle inequality holds: $h(v') \leq k(v', v) + h(v)$ [10]. Here $k(v, v')$ is the distance of the shortest path between v and v' . Since the heuristic is admissible when it is posed with a non-deterministic choice, it picks the most conservative approach even if its not viable. As seen earlier, this leads to an underapproximation of the estimate of distance to the error. In building the ICFG, the FSM heuristic discards a lot of useful calling context information. We address this problem in two parts. First we are going to find a new representation for the program with a bounded calling context. Second we are going to present an extended FSM heuristic that takes the runtime information on the stack of the concrete state and computes the heuristic on the new representation.

3. IMPROVING ERROR DISCOVERY

An interprocedural inlined flow graph (IIFG) is defined in [16] to preserve the syntactic-semantic relationship in an ICFG. In an IIFG all the procedures are inlined at their call sites to build a full context of the procedure call sequence. This graph overcomes the inadequacies of the ICFG used in the FSM distance heuristic as shown in the previous section because it is fully context sensitive. An FSM distance heuristic computed on an IIFG graph is closer to actual distance between the current state of the search and an error in terms of the number of transitions needed to reach that state. There are, however, two obstacles to building an IIFG. The first difficulty is that graphs for recursive programs are infinite. The second difficulty is that the size of the IIFG can be prohibitive in big non-recursive programs that possess a high degree of nested function calls. These two obstacles can be mitigated by not statically constructing, but dynamically reconstructing, the IIFG from the current state of the search. In other words, we use the context information in the runtime stack of the running process to build a partial IIFG. Note, in talking about the runtime stack, we do not mean the runtime stack of model checking process, but the runtime stack of the program (software artifact) in which the model checker is searching for an error state. To dynamically reconstruct the IIFG, we first statically build an ICFG with bounded calling context, and we then traverse it by unwinding the call frames in the runtime stack.

3.1 Augmented Interprocedural Control Flow Graph

An augmented ICFG (AICFG) gives an accurate representation of the calling context in procedure calls up to a bounded stack depth of size k as specified by the user. If k is set to be 0, then we end up with a regular ICFG. If k is set to infinity, then we end up with a full IIFG. Our heuristic balances computation resources in static construction of the AICFG with accuracy in its heuristic estimate of the FSM distance.

Formally, an AICFG is an ICFG, $G = \{V, E\}$, with vertices that not only label a specific location in the control flow graph, but also as many as k return location depending on calling context in the runtime stack. Vertices are thus identified by PC values and a calling context less than or equal to k .

Algorithm: `make_AICFG(abstract_State sa)`
1: `/* V:= {}, E:= {}, G:= {V, E} */`
2: **if** `sa ∈ V` **then**
3: **return**
4: `V := V ∪ {sa}`
5: **for each** `s'a ∈ abs(succ(abs-1(sa)))` **do**
6: `make_AICFG(s'a)`
7: `E := E ∪ (sa, s'a)`

Figure 2: Pseudo Code to build an Interprocedural Control Flow Graph

An AICFG for a graph is created from a depth-first traversal of the program that includes k slots for return addresses in the runtime stack. An algorithm for this is shown in Figure 2. The function abs^{-1} in the algorithm takes a vertex in the AICFG and maps onto a set of real states that might be encountered in the guided search of the program. Its dual function, abs , maps a real state from the guided search onto a single vertex in the AICFG by abstracting away any information other than the PC value and the top k return addresses in the runtime stack. The succ function generates the successors of a set of states. These states are real states that might be encountered in executing the program on its native hardware. Note that the algorithm considers the successors of abstract states in a depth-first manner. All the visited vertices are marked in a hash table. If a successor s'_a is already contained in the hash table, either a cycle exists in the actual IIFG of the program or the calling context is greater than the bounded depth k used in the AICFG construction..

For the purposes of this paper we are going to pick $k = 1$ to simplify the examples; although, this is not a requirement of our algorithm. An AICFG for the simple example presented in Figure 1(a) is shown in Figure 3. The function `foo` is now inlined in the graph due to the added information in the AICFG vertices. The first vertex $\langle 01, (init) \rangle$ has the PC value 01 and indicates that there have been no function calls. Now when `foo` is called at PC 02, the first vertex in `foo` is $\langle 06, RA : 03 \rangle$. This vertex has context sensitive information that it is part of the procedure `foo` which was called from $\langle 02, (init) \rangle$ and will return to $\langle 03, (init) \rangle$. While model checking the program, if the PC value in the current state of the search is 06 and the return address on top of the stack is 03, then we can find the corresponding abstract state, $\langle 06, RA : 03 \rangle$, in the AICFG using the abs function. The FSM distance heuristic now gives an estimate of 8 steps to the error. This is the true distance to the error in this example. Let us consider a slightly more complicated example where this approach fails due to the bounded calling context.

The extended example shown in Figure 4(b) has two levels of function calls, $x \rightarrow f \rightarrow g$ and $y \rightarrow f \rightarrow g$. The procedure `f` is still inlined in the graph, but procedure `g` is at depth 2, which is greater than the specified bound of $k = 1$, it is not inlined in the graph, as seen in Figure 4(a). While model checking, if the current state is in procedure `g`, the heuristic is faces the same non-deterministic choice, as it does on the ICFG. To retain admissibility, it picks the most conservative

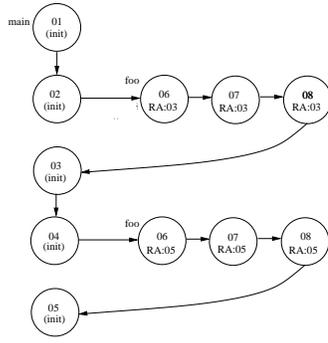


Figure 3: Augmented Interprocedural Control Flow graph for the simple example in Figure 1(b)

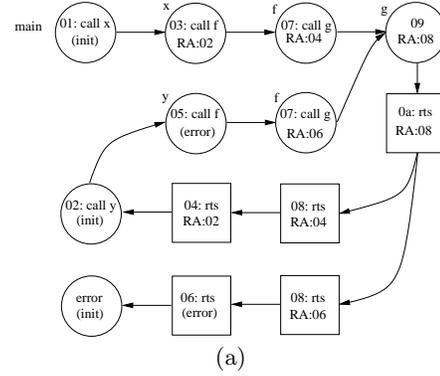
path. This creates the same problem of underestimation, as seen on the ICFG. Also, for a recursive program if we are at a recursive depth greater than k , the underestimation problem persists. To overcome this, we are going to use an extended FSM heuristic that uses the calling context found in the current state of the search to dynamically reconstruct a portion of the IIFG.

The Extended FSM algorithm takes the runtime stack and unrolls it incrementally to compute the heuristic which simulates partially constructing the IIFG. From the state where the stack is fully unrolled, it computes the FSM distance to an error on the AICFG to give a better estimate of the distance to the error. The algorithm to compute the extended FSM heuristic is presented in Figure 5. In this figure *distance* is a set of possible estimates to the error state. The variable d is a counter that keeps track of the heuristic estimate. The example presented in Figure 4 will be used to demonstrate the algorithm.

3.2 Extended FSM Distance Heuristic

The function *aicfgState* in Figure 5 generates abstract states based on the PC value and the return addresses in the runtime stack of the current state in the guided search. Figure 6(a) visualizes the strategy for extracting the abstract states. For $k = 1$ the first abstract state consists of the PC value and the return address on top of the stack $\langle 09, RA : 08 \rangle$. This indicates when a return statement (*rts*) is encountered in the current subroutine, it will transfer flow of execution to a location in the program where the PC is 08. The next return address on the runtime stack points to the preceding call site of the program. On the next *rts* instruction, the control will jump to location 04 in the program. Hence the next abstract state is $\langle 08, RA : 04 \rangle$. This process keeps repeating till the stack is exhausted and the last abstract state $\langle 02, (init) \rangle$ is generated. The algorithm in Figure 5 uses a Worklist to store these abstract states (line 3). In general terms, for an arbitrary k bound the abstract state looks at k entries on the stack.

The algorithm iterates through the Worklist till it is empty. During each iteration, an abstract state (s_a) is removed from the Worklist (line 5). For each abstract state the function *return_statement* returns another abstract state (s_{rts}) that corresponds to the return statement of the pro-



main:		f:	
01: call x		07: call g	
02: call y		08: rts	
error			
x:	y:	g:	
03: call f	05: call f	09: xyz	
04: rts	06: rts	0a: rts	

(b)

Figure 4: An extended example (a) An ICFG for a program with nested functions (b) Program with nested functions

Algorithm: *Extended_FSM*(state S)

```

1: /*  $d := 0$  */
2: /*  $distance := \{\}$  */
3: Worklist := aicfgState( $S$ )
4: while (Worklist) do
5:   Remove  $s_a$  from Worklist
6:    $s_{rts} := returnStatement(s_a)$ 
7:   if error_postdominates( $s_a$ ) then
8:      $x := d + FSM(s_a, err)$ 
9:      $distance := distance \cup x$ 
10:    break
11:  else if in_Scope_error( $s_a$ ) then
12:     $x := d + FSM(s_a, err)$ 
13:     $distance := distance \cup x$ 
14:     $d := d + FSM(s_a, s_{rts}) + 1$ 
15: return min(distance)

```

Figure 5: Pseudo Code for the Improved ICFG Algorithm

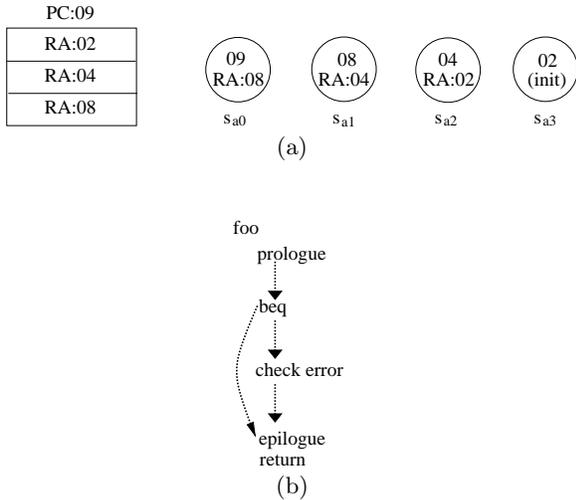


Figure 6: Understanding the Improved ICFG Algorithm (a) Abstract states generated from the runtime stack (b) Control flow of a subroutine with an error

cedure containing s_a (line 6). The abstract states representing the return statements of procedures are marked statically while building the AICFG. For the abstract state, $\langle 09, RA : 08 \rangle$, which is in procedure g the corresponding s_{rts} is $\langle 0a, RA : 08 \rangle$.

The first condition in the algorithm checks whether the error postdominates s_a or not (line 7). In essence, it is checking if all paths from s_a to s_{rts} pass through the error state. If it does, there is no need to iterate through the rest of the abstract states. We compute the FSM distance from s_a to the error state, add it to the heuristic counter (line 8), and append it to the set of possible estimates (line 9). We break from our iteration (line 10) and return the lowest estimate in the set *distance* (line 15).

A second check is performed to see if the error is in scope of the current procedure. If there exists a path from s_a to the error which does not include s_{rts} , then the error is said to be in scope. Consider the simple flow diagram in Figure 6(b), the error is control dependent on a conditional branch. Based on which branch is taken we can either branch to the error vertex or jump to the epilogue of the function. In such a case we need to consider two options. Either the error can be reached in the current procedure, or unrolling the loop discovers a shorter path to the error. If the error is in scope, we take the shortest FSM distance amongst paths from s_a to the error that precludes the return statement, add it to the heuristic counter (line 12), and append this heuristic value to the set of possible estimates to the error (line 13).

Finally, if the error does not postdominate s_a , then we compute the FSM distance from s_a to s_{rts} , add 1 to this value, to account from the outgoing edge from s_{rts} to the *return* vertex, and then increment the heuristic counter by this value (line 14). After we finish iterating through the Worklist, we return the smallest member of the *distance* set.

Let us consider a concrete example of how the algorithm works. Given the extended example Figure 4(b), let the current state be represented by the left side of Figure 6(a). Based on the abstract states generated for the current state the first s_a is $\langle 09, RA : 08 \rangle$, and the corresponding s_{rts} is $\langle 0a, RA : 08 \rangle$. There is no error in this procedure, the two checks on postdominance and scoping fails. The FSM distance between s_a and s_{rts} is 1, we add an additional 1 to it for the outgoing egde to the *return* vertex, so the value of d is set to 2. The next abstract state processed is $\langle 08, RA : 04 \rangle$. The s_a and the s_{rts} are the same in this case. So the FSM distance will be 0 but we will add 1 to d to account for the outgoing edge from the *exit* point. The value of d is now 3. The above is process is repeated till we get to the last abstract vertex $\langle 02, (init) \rangle$ and d is 4 at this point. The error now postdominates the s_a . The FSM distance from $\langle 02, (init) \rangle$ to the error is 7. This value is added to the existing d and appended to the distance set. At this point we break out of the loop, since it is the last vertex in the Worklist. For this particular example, the distance set just has a single element, so we return that element. For this particular example the estimate is 11, which is also the true distance to the error.

To calculate the heuristic estimate for a concurrent program with multiple processes, the FSM distance heuristic presented in [12] sums the distance from the current PC value to the error state for every process. We calculate the heuristic estimate based on what property of the concurrent program is being verified. A *mutex* violation occurs if two or more threads are in the critical section. The heuristic estimate for a *mutex* violation is the sum of distances in two processes which have the shortest paths to the critical section compared to other processes. Now consider the property which is a check to see if any of the processes reach 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 process. Another useful property checked in concurrent programs is whether two or more processes are *deadlocked*. In this case we take the summation of the distances from the current state to the error state. For a *deadlock* condition of two processes in a three process system the sum of the two shortest distances from the set of estimated distances for all threads is returned as the heuristic estimate. And if we are looking at a *deadlock* state for all processes then we take the the summation of distances from the current state to the error state for all threads. This is also the default property specified in a concurrent program if the user does not specify any specific property.

The extended FSM distance algorithm can be used on any flow graph. It will accurately dynamically reconstruct a part of the IIFG on both, an ICFG, and an AICFG. Now the question becomes: why do we need an AICFG? The extended FSM distance algorithm is limited by the information on the runtime stack. It can simulate construction only part of the IIFG based on the return addresses on the runtime stack. Once the call stack is fully unrolled, the heuristic estimate is dependent on the accuracy of the control-flow representation. Consider the example Figure 4(b), the extended heuristic generates the IIFG upto the vertex $\langle 02, (init) \rangle$. From this vertex to the error, the FSM distance is computed

and added to the estimate. Suppose there was another procedure *baz* which was called twice after vertex $\langle 02, (init) \rangle$ and before the error. On the ICFG, the return from *baz* will have two outgoing edges and while calculating the FSM distance, the most conservative path will be picked. This will lead to the same underestimation we encountered earlier. If we compute the heuristic on an AICFG, depending on our value of k , we will get a better estimate of the distance to the error.

To make this algorithm efficient, and not recompute the heuristic with the same stack values, we mark certain vertices in the graph. For the current and abstract state values shown in Figure 6(a), we calculate the heuristic value of 11 in Figure 4(a). Now if we change the PC from 09 to 0a, and retain the same return addresses on the stack on the left side of Figure 6(a). Among the abstract states on the right side of Figure 4(a), only s_{a0} will change to $\langle 0a, RA : 08 \rangle$. In such a case we do calculate the whole heuristic estimate again. In the abstract states in Figure 4(a), which represent the return points of a procedure, we will add a table of heuristic estimates mapped to particular stack values. These states are annotated based on the contents of the runtime stack. The s_{rts} for procedure *g* is marked $(08, 04, 02 := 10)$. The s_{rts} for procedure *f*, $\langle 08, RA : 04 \rangle$ is marked $(04, 02 := 9)$. In the extended FSM algorithm before unrolling the runtime stack further at any abstract return vertex (s_{rts}), an additional check is performed. The annotated table entries in the abstract return vertices are compared against the runtime stack values. If a match is found, then the algorithm increments the heuristic estimate by the distance marked in the table entry and terminates. For example, if the values on the runtime stack are $(04, 02)$ and we encounter a s_{rts} vertex marked $(04, 02 := 9)$, we simply add 9 to the heuristic estimate and terminate. By simply marking the s_{rts} , the size of the AICFG does not increase, and it is a good spot to check for corresponding stack values.

The extended FSM distance heuristic is admissible and consistent. The AICFG is a more refined representation of the program based on the depth of the calling context. Hence the proof for admissibility and consistent follows the one presented in [12]. The infeasible paths in the AICFG encountered after the unrolling of the stack will always underestimate the distance to the error and this does not violate admissibility.

4. RESULTS

To measure the improvement in the heuristic estimate we compare the extended FSM distance heuristic to the regular FSM distance heuristic. We also compare it to other exhaustive searches like BFS and DFS. The Hamming distance performs poorly with respect to the FSM distance so it is omitted in this comparison [12]. Tests were performed on a set of benchmarks developed for the gnu debugger based model checker Estes [21]. The set of benchmarks consist of models with concurrency errors. This heuristic can be implemented in any model checker where it is possible to derive a control flow representation of the program being model checked.

The results from a Pentium III 1.8 Ghz processor with 1 GB of RAM are shown in Table 1 and Table 2. These were run

on Estes, with a 6.1.1 version of the gnu debugger, using the m68hc11 backend simulator. In Table 1 we show the total number of states enumerated before finding the error state. In Table 2 we measure total wall clock time from the start of the program till the error state is found. We measure this using the UNIX utility *time*.

The extended FSM heuristic outperforms the regular FSM heuristic by generating fewer number of states in our benchmarks. An initial improvement is seen in the reduction of states generated by the removal of false edges caused by indirect jumps and subroutine calls. A more significant decrease in the number of states generated is achieved by implementing the extended FSM distance heuristic. In some cases the extended FSM heuristic is faster in finding the error in terms of total time taken to find the error. There is an overhead of building the AICFG, extracting the runtime stack, dynamically reconstructing the IIFG, and computing the heuristic on it. This causes the total running time to increase in some of the models. For all testing purposes the bound of k was chosen to be 1 in the AICFG.

In the Hyman model there are not too many procedure calls. By just resolving indirect jumps *a priori* the number of states generated reduced to 1500 and the implementation of the extended FSM distance heuristic caused this number to drop to 881. The next few models are based on the dining philosophers problem which is often used in benchmarking guided search heuristics for model checking. Figure 7 demonstrates the call structure of the basic program used. The different versions of *dining philosophers* have a varying degree of non-determinism and different numbers of threads. When a thread is created it keeps iterating through the *dining* function, where it makes function calls to *sleep*, *getleftfork*, *checkfordeadlock*, and *getrightfork*. If all the philosophers hold a single left fork, none of them will release their fork until they get a right fork. This leads to a state of starvation. This is defined as the error state for the model. The Naive Dining-Phil always wants to eat and sleep. In Dining-Phil the philosophers make a non-deterministic choice whether they want to sleep, eat, or release forks. The Moody Dining-Phil in addition to the choices of Dining-Phil decide the length of time they want to sleep. As demonstrated earlier in the regular ICFG, the calling context is missing in the representation. The multiple function calls in the program add a lot of non-deterministic branches to the graph. This causes the FSM distance to heuristic underestimate. In three of the benchmark examples, the FSM distance heuristic takes the maximum number of states to find the error. In the Naive Dining-Phil3 FSM distance heuristic generates 1.25 times more states than the BFS. This goes to show that for some models the FSM distance generates into a random search.

Estes is very flexible in terms of fine grain control of thread scheduling. For the philosophers with two threads, it is allowed to preempt at assembly level instructions anywhere in the *dining* and the *sleep* functions. The Dining-Phil2 and Moody Dining-Phil2 have a greater branching factor in the transition graph, and a higher degree of non-determinism. The extended FSM distance heuristic generates five times fewer states than its closest competitor in these examples. Generating fewer states mostly compensates for the over-

reaches either a *read* instruction or an entry point of the procedure, it cannot statically resolve the feasibility of the path. We can first prune the infeasible paths statically and then at *read* statements, and at beginning of procedures, perform the infeasible path analysis again during model checking. By combining static and dynamic analysis, we hope to prune a lot of infeasible paths in the program. This can lead to a better estimate of the distance to the error. We also need to combine property based heuristics with structural heuristics. For example, if a certain variable is part of the property being verified, then we can check how far we need to go before its value is changed. This can be used in tandem with the structural heuristic which gives us the distance between two points.

6. REFERENCES

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.
- [2] T. Ball and S. Rajamani. The SLAM toolkit. In G. Berry, H. Comon, and A. Finkel, editors, *13th Annual Conference on Computer Aided Verification (CAV 2001)*, volume 2102 of *Lecture Notes in Computer Science*, pages 260–264, Paris, France, July 2001. Springer-Verlag.
- [3] T. Ball and S. K. Rajamani. Bebop: A symbolic model checker for boolean programs. In K. Havelund, J. Penix, and W. Visser, editors, *7th International SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer, August 2000.
- [4] R. Bodik, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. *SIGSOFT Softw. Eng. Notes*, 22(6):361–377, 1997.
- [5] D. Callahan. The program summary graph and flow-sensitive interprocedural data flow analysis. In *PLDI '88: Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 47–56, New York, NY, USA, 1988. ACM Press.
- [6] E. Clarke and D. Kroening. Hardware verification using ANSI-C programs as a reference. In *Proceedings of ASP-DAC 2003*, pages 308–311, Yokohama City, Japan, January 2003. IEEE Computer Society Press.
- [7] E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2004)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, Barcelona, Spain, April 2004. Springer.
- [8] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *International Conference on Software Engineering*, pages 37–46, 2001.
- [9] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, R. Zheng, and H. Zheng. Bandera: extracting finite-state models from java source code. In *International Conference on Software Engineering*, pages 439–448, 2000.
- [10] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of a*. *J. ACM*, 32(3):505–536, 1985.
- [11] S. Edelkamp, A. Lluch-Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *8th International SPIN Workshop on Model Checking Software*, number 2057 in *Lecture Notes in Computer Science*. Springer-Verlag, 2001.
- [12] 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.
- [13] P. Godefroid. Software model checking: The VeriSoft approach. Technical report, Bell Laboratories, Lucent Technologies, 2003.
- [14] S. Graf and L. Mounier, editors. *Model Checking Software: 11th International SPIN Workshop*, volume 2989 of *Lecture Notes in Computer Science*, Barcelona, Spain, April 2004. Springer.
- [15] A. Groce and W. Visser. Model checking java programs using structural heuristics. In *2002 ACM SIGSOFT International symposium on software testing and analysis*, 2002.
- [16] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *1998 ACM SIGSOFT International symposium on software testing and analysis*, 1998.
- [17] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994.
- [18] T. A. Henzinger, R. Jhala, R. Majumdar, , and G. Sutre. Software verification with Blast. In T. Ball and S. Rajamani, editors, *Proceedings of the Tenth International Workshop on Model Checking of Software (SPIN)*, volume 2648 of *Lecture Notes in Computer Science*, pages 235–239, Portland, OR, May 2003.
- [19] G. J. Holzmann and R. Joshi. Model-driven software verification. In Graf and Mounier [14], pages 76–91.
- [20] T. Mehler and S. Edelkamp. Directed error detection in C++ with the assembly-level model checker StEAM. In Graf and Mounier [14], pages 39–56.
- [21] E. G. Mercer and M. Jones. Model checking machine code with the gnu debugger. In *12th International SPIN Workshop*, San Francisco, USA, August 2005. Springer.
- [22] N. Nethercote. *Dynamic Binary Analysis and Instrumentation*. PhD thesis, Computer Laboratory, University of Cambridge, United Kingdom, Sept. 2004.

- [23] N. Nethercote and J. Fitzhardinge. Bounds-checking entire programs without recompiling. In *Informal Proceedings of the Second Workshop on Semantics, Program Analysis, and Computing Environments for Memory Management (SPACE 2004)*, Venice, Italy, Jan. 2004.
- [24] N. Nethercote and J. Seward. Valgrind: A program supervision framework. In O. Sokolosky and M. Viswanathan, editors, *Proceedings of the Third Workshop on Runtime Verification (RV'03)*, volume 89 of *Electronic Notes in Theoretical Computer Science*, pages 1–23, Boulder, CO, USA, July 2003. Elsevier.
- [25] C. Pasareanu and W. Visser. Verification of Java programs using symbolic execution and invariant generation. In Graf and Mounier [14], pages 164–181.
- [26] 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)*, pages 488–497, Limerick, Ireland, June 2000. ACM.
- [27] Robby, M. B. Dwyer, and J. Hatcliff. Bogor: An extensible and highly-modular model checking framework. *ACM SIGSOFT Software Engineering Notes*, 28(5):267–276, September 2003.
- [28] K. Seppi, M. Jones, and P. Lamborn. Guided model checking with a bayesian meta-heuristic. *Fundamenta Informatica*, To Appear, 2005.
- [29] J. Seward and N. Nethercote. Using valgrind to detect undefined value errors with bit-precision. In *Proceedings of the USENIX'05 Annual Technical Conference*, Anaheim, California, USA, Apr. 2005.
- [30] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda. Model checking programs. *Automated Software Engineering Journal*, 10(2), April 2003.
- [31] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, pages 131–144. ACM Press, 2004.
- [32] C. Yang and D. Dill. Validation with guided search of the state space. In *35th Design Automation Conference (DAC98)*, pages 599–604, 1998.