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

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. In these examples, 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: Guided search, structural heuristics

General Terms: Verification, algorithms, reliability

1. INTRODUCTION

Explicit state model checking is a verification method that proves whether a transition system can or will reach an error state that violates a pre-defined property. For example, the Java Pathfinder tool model checks the actual software using a Java virtual machine [10]. Similar approaches use simulators and debuggers for other machine architectures [7, 8]. These approaches retain a high-fidelity model of the target execution platform with low-level control of scheduling decisions.

The primary challenge to explicit model checking is managing the size of the transition system. For large software systems, the computational resources are quickly exhausted before model checking finishes exploration. One solution to state explosion is guided model checking. Guided model checking focuses on error discovery by using heuristics to prioritize the search. The idea is to discover an error before computational 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 [11]. Hamming distance heuristics ignore the property being verified as well as the structure of the transition system. The property being verified is accounted for in [2]. The approach is further refined with Bayesian reasoning in [9]. These heuristics only consider the state representation and the property being verified.

Guided search heuristics can be improved by considering program structure together with the property being verified. Heuristics related to Java programs are described in [4]. Finite state machine (FSM) distance heuristics in [3, 1] exploit the program structure by extracting the control flow of the program to compute a heuristic estimate. The FSM distance heuristic is not context sensitive; this means that it ignores the calling context of functions. The lack of 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. 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 improve the estimate of the distance to an error state. Improvement 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 structure and its flow of execution. Edelkamp and Mehler in [3] 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)[6]. Hence we will refer to the graph in [3] as an ICFG. The FSM distance is defined as the minimal number of operations required to reach an error state from the current state. 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 ICFG vertex for the error from the current vertex in the ICFG. It returns this length as the heuristic estimate to guide the search. The FSM heuristic is admissible and consistent.

ASE'05, November 7–11, 2005, Long Beach, California, USA.

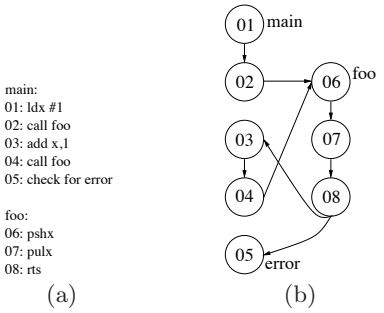

Figure 1: A simple example (a) A program that calls foo twice and checks for error (b) The control flow graph indexed on PC value

The work in this paper builds on the FSM distance heuristic. We demonstrate the algorithm and also its shortcomings. The assembly instructions of a simple C program, where procedure `foo` is called twice from procedure `main`, are shown in Figure 1(a). The assembly-level representation of the program is used to obtain the finest granularity of the program. Errors arising due to data inconsistencies and scheduling techniques can be detected at this level as shown in [8]. Figure 1(b) is the ICFG for the program. In building the ICFG we assume indirect jumps only target entries in defined jump tables, and that indirect procedures calls only target valid entry points in procedures. For convenience, we label the vertex in the ICFG with the program counter (PC) values from the corresponding locations indicated to the left of the program. These PC values are unique labels for each line of the source program. Let us suppose that the current state of the program is at PC value 06 in the explicit state 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 value is returned as the heuristic estimate for this current state. When the search is in the function `foo`, the return address on the runtime stack can be either 03 or 05 for the corresponding call sites 02 and 04 in `main`. If the actual call site is from line 02 in the program, then the return address is 03; thus, the FSM heuristic uses an infeasible path and underestimates the distance, given the current state.

3. IMPROVING ERROR DISCOVERY

We present a new heuristic to facilitate improved error discovery. Our technique consists of two parts. First, we refine the control flow representation to include context sensitive information for a certain bound. Second, we extend the FSM distance heuristic to dynamically prune the infeasible paths in the control flow representation by using the information on the runtime stack

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 obtain a regular ICFG. If k is set to infinity, then we obtain a interprocedural inlined flow graph (IIFG) [5]. An IIFG preserves the syntactic-semantic

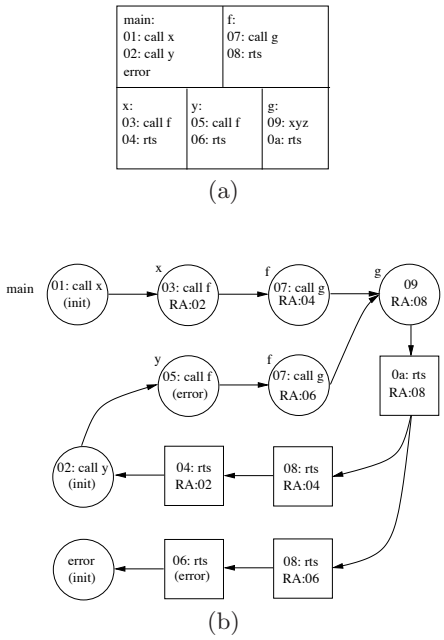

Figure 2: An extended example (a) Program with nested functions (b) An AICFG for a program with nested functions

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. The FSM distance computed on an IIFG is more accurate in its distance estimate; however, an IIFG can be prohibitive in size for large programs with a high degree of nested function calls. Our heuristic balances computation resources in static construction of the AICFG with accuracy in its heuristic estimate of the FSM distance. The vertices in the AICFG are labeled with PC values, and k return locations on the call stack.

The extended example shown in Figure 2(a) has three levels of function calls, $main \rightarrow x \rightarrow f \rightarrow g$ and $main \rightarrow y \rightarrow f \rightarrow g$. For the purposes of this paper, we pick $k = 1$ to simplify the presentation; although, this is not a requirement of our algorithm. An AICFG for the extended example is presented in Figure 2(b). The procedure `f` is at depth $k = 2$, but since procedures `x` and `y` are called only once, the procedure `f` along with `x` and `y`, is fully inlined in the graph. This means that procedures `f`, `x`, and `y` have the context to be mapped to their distinct call sites. The vertex: $\langle 07, RA : 04 \rangle$ in the AICFG has context sensitive information that it was called from PC value 03 and returns to PC value 04 in procedure `x`. Procedure `g` is at depth 3, which is greater than the specified bound of $k = 1$. Additionally its predecessor, procedure `f`, is called more than once; hence, procedure `g` is not inlined in the graph, as seen in Figure 2(b). While model checking, if the current state is in procedure `g`, then there is no context sensitive information in the AICFG about which procedure called `g`. To retain admissibility, the FSM distance heuristic computed on an AICFG picks the most conservative path when posed with a non-deterministic choice. This creates the same problem of underestimation, as seen on the ICFG. We overcome this problem by using information on the runtime stack which is

```
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 := return_statement(s_a)
 7:    if error_postdominates(s_a) then
 8:       x := d + FSM(s_a, error)
 9:       distance := distance ∪ {x}
10:       break
11:    if in_scope_error(s_a) then
12:       x := d + FSM(s_a, error)
13:       distance := distance ∪ {x}
14:    d := d + FSM(s_a, s_rts) + 1
15: return min(distance)
```

Figure 3: Pseudo Code for the Improved ICFG Algorithm

availabe in the state generated during model checking becuase the entire runtime stack of the program is a part of the state. Our approach extracts the runtime stack from the current state and unrolls it incrementally to simulate partially constructing the IIFG to compute the heuristic. From the state where the stack is fully unrolled, it computes the FSM distance from that state to the error using the AICFG and adds it to the heuristic. This gives a better estimate of the distance to the error. The algorithm to compute the extended FSM heuristic is presented in Figure 3. The example shown in Figure 2 is used to demonstrate the algorithm.

The variable *distance* is a set of possible estimates to the error state, and variable d is a counter that keeps track of the heuristic estimate in Figure 3. The function aicfgState generates AICFG vertices from the PC value and return addresses on the runtime stack of the current state in the model checker.

Figure 4(a) visualizes the strategy for extracting the AICFG vertices from a current state. The left side of Figure 4(a) represents the values of the current state, and the right side shows the AICFG vertices generated by the function aicfgState for those values. For $k = 1$ the first AICFG vertex concatenates the PC value and the return address on top of the stack $\langle 09, RA : 08\rangle$. This indicates that when a return statement is encountered in the current subroutine, it transfers flow of execution to a location in the program where the PC value 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 jumps to PC value 04 in the program. Hence the next AICFG vertex is $\langle 08, RA : 04\rangle$. The function, aicfgState returns when the stack is exhausted and the last AICFG vertex $\langle 02, (init)\rangle$ is generated. In general terms, for an arbitrary k bound, the AICFG vertex is derived from the first k return locations on the runtime stack and the PC. We first build the AICFG statically, and then start model checking the program. For a current state generated during model checking, the aicfgState function generates a set of AICFG vertices that represent a path from the start of the program to the current state. The algorithm uses a Worklist to store these AICFG vertices (line 3). We assume there is a single return point for each procedure. Experience has shown this assumption to be true for most compilers.

The algorithm iterates through the Worklist till it is empty. During each iteration, an AICFG vertex (s_a) is removed

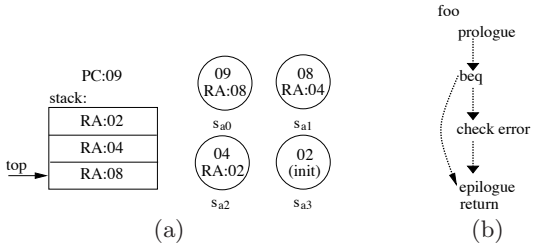

Figure 4: Understanding the Improved ICFG Algorithm (a) AICFG vertices generated from the runtime stack (b) Control flow of a subroutine with an error

from the Worklist (line 5). For each AICFG vertex, the function return_statement returns another AICFG vertex (s_{rts}) that corresponds to the return statement of the procedure containing s_a (line 6). The AICFG vertices representing the return statements of procedures are marked statically while building the AICFG. For the AICFG vertex $\langle 09, RA : 08\rangle$, which is in procedure `g` in Figure 2(b), 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 checks 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 AICFG vertices. We compute the FSM distance from s_a to the error state, add to 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 4(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 the two options. First, the error can be reached in the current procedure, second unrolling the runtime stack further 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. Otherwise regardless of whether the error is in scope, we compute the FSM distance from s_a to s_{rts}, add 1 to this value, to account for the outgoing edge from s_{rts} to the *return* vertex, and then increment the heuristic counter by this value (line 14). The processing of the last AICFG vertex computes the FSM distance on the AICFG between the AICFG vertex, at the top most level of the calling structure, and the error state. So even if the error is not in scope of the procedures on the runtime stack, we find a path to the error, assuming the error is reachable from the `main` procedure. 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. For the extended example in Figure 2(a), let the current state be represented by the left side of Figure 4(a). Based on the AICFG vertices generated for the current state, the first s_a is $\langle 09, RA : 08\rangle$, and the corresponding s_{rts} is $\langle 0a, RA : 08\rangle$. The error node is not part of the current

Table 1: States generated before error and time taken to find the error

	Breadth First Search	Depth First Search	FSM distance	Extended FSM distance
Hyman	3,550 : 1.70s	5,944 : 2.89s	2,715 : 1.33s	881 : 1.28s
Naive Dining-Phil2	19,013 : 7.33s	8,066 : 2.02s	22,701 : 13.38s	3,155 : 3.56s
Dining-Phil2	48,068 : 20.56s	33,523 : 10.52s	87,974 : 53.65s	6,535 : 6.95s
Moody Dining-Phil2	87,974 : 40.62s	33,523 : 10.09s	86,139 : 51.42s	6,535 : 7.01s
Naive Dining-Phil3	485,648 : 3m23s	382,359 : 1m48s	608,595 : 5m52s	369,328 : 9m42s
Naive Dining-Phil3.1	18,281 : 9.45s	10,667 : 2.93s	12,674 : 8.69s	9,341 : 14.62s
Dining-Phil3.1	77,777 : 37.88s	75,947 : 22.58s	40,327 : 26.81s	34,328 : 52.60s
Moody Dining-Phil3.1	118,979 : 59.44s	68,233 : 20.33s	51,163 : 34.83s	40,749 : 1m3s

procedure, so the two checks on postdominance and scoping fail. The FSM distance between s_a and s_{rts} is 1, we add an additional 1 to it for the outgoing edge to the return vertex, so the value of d is set to 2. The next AICFG vertex processed is $\langle 08, RA : 04\rangle$. The s_a and the s_{rts} are the same in this case. The FSM distance is 0, but we add 1 to d to account for the outgoing edge from the return points. The value of d is now 3. We iterate through the loop until we get to the last AICFG vertex, $\langle 02, (init)\rangle$, and the value 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. The distance set has a single element of value 11 that is returned as the heuristic estimate. The estimate is the true distance in this case.

The extended FSM distance algorithm can be used on any flow graph. It dynamically reconstructs 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 construct 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 in Figure 2(a), the extended heuristic generates the IIFG up to the vertex $\langle 02, (init)\rangle$. From this vertex to the error, the FSM distance is computed directly on the AICFG without further context information. 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 node from `baz` will have two outgoing edges; thus, while calculating the FSM distance, the most conservative path will be picked. This leads to the same underestimation shown earlier. If we compute the heuristic on an AICFG, depending on our value of k, we get a better estimate of the distance to the error. The extended FSM distance heuristic is admissible and consistent. Its proof follows the one presented in [3]. The AICFG is a more refined representation of the program based on the depth of the calling context. The infeasible paths in the AICFG encountered after the unrolling of the stack will always underestimate the distance to the error and this do not violate admissiblity and consistency.

4. RESULTS AND CONCLUSIONS

The results from a Pentium III 1.8 Ghz processor with 1 GB of RAM are shown in Table 1. These were run on Estes [8], with a 6.1.1 version of the gnu debugger using the m68hc11 back-end simulator. We show the total number of states enumerated before finding the error state, and we measure total wall clock time from the start of the programs till the error state is found. Each cell in the table has the format $\langle Number\ of\ states\rangle : \langle Time\ taken\rangle$. The tests are performed on a set of benchmarks consisting of models with concurrency errors. For all testing purposes the bound of k was chosen to be 1 in the AICFG.

The extended FSM heuristic outperforms the regular FSM heuristic by generating fewer number of states in our benchmarks before error discovery. 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. In large systems, this overhead is acceptable given we find an error before the memory runs out.

5. REFERENCES

[1] 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.

[2] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proceedings of the 7th International SPIN Workshop*, number 2057 in Lecture Notes in Computer Science. Springer-Verlag, 2001.

[3] S. Edelkamp and T. Mehler. Byte code distance heuristics and trail direction for model checking Java programs. In *Workshop on Model Checking and Artificial Intelligence (MoChArt)*, pages 69–76, 2003.

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

[5] M. J. Harrold, G. Rothermel, and S. Sinha. Computation of interprocedural control dependence. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 11–20, New York, NY, USA, 1998. ACM Press.

[6] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Trans. Program. Lang. Syst.*, 16(2):175–204, 1994.

[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*, volume 2989 of *Lecture Notes in Computer Science*, pages 39–56. Springer, 2004.

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

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

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

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