Heuristic Model Checking for Java Programs

Alex Groce[1] and Willem Visser[2]

[1] School of Computer Science, Carnegie Mellon University
agroce+@cs.cmu.edu
[2] RIACS/NASA Ames Research Center
wvisser@riacs.edu

1 Introduction

Two recent areas of interest in software model checking are checking programs written in standard programming languages [1, 5] and using heuristics to guide the exploration of an explicit-state model checker [3]. Model checking real programs has the drawback that programs often contain a larger degree of detail than designs and hence are more difficult to check (due to the more acute state-explosion problem); however the large amount of detail in a program allows more precise heuristics for narrowing down the search space when using a model checker for error-detection. This paper describes the addition of support for heuristic (or directed) search strategies to Java PathFinder (JPF), an explicit state model checker for Java bytecode that uses a custom-made Java Virtual Machine (JVM) [5].

The primary benefits of heuristic search are: discovery of errors that a depth-first search fails to find, and shorter (and thus easier to understand and correct) counterexample paths. In JPF a number of pre-defined heuristics are provided, but a user can also write their own by using an interface to the JVM. The rest of the paper is structured as follows: in section 2 we illustrate JPFs heuristic capabilities by showing two novel predefined heuristics as well as a simple user-defined heuristic, in section 3 we give results of using JPF with heuristics and in section 4 be briefly sketch some future work.

2 Search Capabilities

The heuristic search options in JPF are primarily aimed at checking for deadlocks and assertion violations. Using heuristics for more general LTL properties is possible, but complicates the search strategy (heuristic searches are not very effective for cycle detection[4]). We have found that two kinds of heuristics work well for a variety of programs in searching for assertion violations and deadlocks.

Branch Exploration: Traditional branch coverage metrics measure the percent of branches in a program covered by test cases. Although using a heuristic that greedily searches for high branch coverage over paths (or globally) performs poorly, something more complex works well: (1) States covering a previously untaken branch receive the best heuristic value. (2) States that are not reached by taking a branch receive the next best heuristic value. (3) States that cover a branch already taken are ranked according to how many times that branch has been taken.

Thread Interleaving: Another useful heuristic is to maximize the interleaving of threads along each path in order to find deadlocks or race conditions that cause null pointer exceptions or assertion violations. Many JVMs and JIT compilers only switch between threads at explicit yields inserted in the code or after multiple bytecodes have been executed (for obvious efficiency reasons). This scheduling can leave errors based on subtle interleavings undetected until the code is used in a different setting. By context switching as much as possible, the interleaving heuristic uncovers some of these subtle errors.

What makes these two heuristics particularly interesting is that they focus on the structure of the program being analyzed: the interleaving heuristic will only work well if a program is concurrent whereas the branch-exploration heuristic is best suited to programs where nondeterministic actions are explored (nondeterminism is most often encoded in a branching control structure, such as an if or case statement). These observations are supported by the results shown in the next section. JPF also includes other heuristics: maximizing the number of blocked threads out of running threads (for deadlock detection), randomized exploration, and counting executions of every bytecode rather than branches. Other options include using the sum of two heuristics, limiting the size of the search queue[1] (sacrificing completeness for focused exploration), and using an A^* search rather than the default best-first search. Users may also change the code in the file UserHeuristic.java to create their own heuristics. Consider a program with a class `Main` with a static field `buffer`, itself an object of a class with integer fields `current` and `capacity`. Following is the code for a heuristic returning either (`capacity` − `current`) or a default value (defined in the UserHeuristic class) if the `Main.buffer` field hasn't been initialized:

```
public int heuristicValue() {
  Reference m = getSystemState().getClass("Main");
  if (m != null) {
    Reference b = m.getObjectField("buffer");
    if (b != null) {
      int current = b.getIntField("current");
      int capacity = b.getIntField("capacity");
      if (current > capacity)
        return 0;
      return (capacity-current);
  } }
  return defaultValue;
}
```

Note that lower heuristic values are better, with zero (0) being the top priority, and negative values indicating that a state should not be queued for exploration (for trimming the state space when a runtime computation can show that no successors of a state can give rise to a counterexample). Options allow states with the same heuristic value to be explored in the order they were generated

[1] Heuristic search in JPF is implemented by generating all successor states from the current state and then adding them to a priority queue depending on the heuristic(s) being used.

(for BFS-like behavior when all states have the same value), in the reverse of the order in which they were generated (for DFS-like behavior), or in the order of their hash values. The code above would be useful if errors were suspected to occur when the buffer was at or near its capacity.

3 Experimental Results

DEOS					
Search/Heuristic	Time	Memory	States Explored	Length	Max Depth
branch	60	91	2,701	136	139
branch(A^*)	59	92	2,712	136	139
statement	62	88	2,195	136	137
statement(A^*)	63	94	2,383	136	137
BFS	FAILS	-	18,054	-	135
DFS	FAILS	-	14,678	-	14,678
DFS (depth 500)	6,782	383	392,479	455	500
DFS (depth 1000)	2,222	196	146,949	987	1,000
DFS (depth 4000)	171	270	8,481	3,997	4,000

Dining Philosophers						
Search/Heuristic	Size	Time	Memory	States Explored	Length	Max Depth
branch	8	FAILS	-	374,991	-	41
BFS	8	FAILS	-	436,068	-	13
DFS	8	FAILS	-	398,906	-	384,286
DFS (depth 100)	8	FAILS	-	1,357,596	-	100
DFS (depth 500)	8	FAILS	-	1,354,747	-	500
DFS (depth 1000)	8	FAILS	-	1,345,289	-	1,000
DFS (depth 4000)	8	FAILS	-	1,348,398	-	4,000
interleaving	8	FAILS	-	487,942	-	16
interleaving (queue 5)	8	2	1	1,719	66	66
interleaving (queue 40)	8	5	5	16,569	66	66
interleaving (queue 160)	8	12	27	62,616	66	66
interleaving (queue 1000)	8	60	137	354,552	67	67
interleaving (queue 5)	16	4	5	6,703	129	129
interleaving (queue 40)	16	16	45	69,987	131	131
interleaving (queue 160)	16	60	207	290,637	131	132
interleaving (queue 1000)	16	FAILS	-	858,818	-	41
interleaving (queue 5)	32	11	32	25,344	257	257
interleaving (queue 40)	32	FAILS	-	472,022	-	775
interleaving (queue 160)	32	FAILS	-	494,043	-	86
interleaving (queue 5)	64	59	206	101,196	514	514

The first results[2] are from the DEOS real-time operating system case study [5], where a very subtle error exists that can lead to an assertion violation. Because the error results from a particular choice of action at a particular point in time on the part of threads, the branch coverage based heuristics (and statement coverage heuristics) find the bug quickly by exercising different options as quickly as possible. The thread interleaving heuristics were not used as the real-time constraints prevent any thread interleaving choices in DEOS. Note how the limited depth DFS searches find much longer counterexamples in each case.

[2] All results obtained on a 1.4 GHz Athlon with JPF limited to 512Mb. Times are in seconds and memory is in megabytes. FAILS indicates failure due to running out of memory.

The second table shows the results of applying heuristics to the classic dining philosophers problem. Here, branch based heuristics are not very successful (since there are almost no branches in the program), but the interleaving-based heuristic plus queue size limitation (analogous to the depth limitations for DFS) produces counterexamples even for quite large numbers of philosophers (threads).

4 Conclusions and Future Work

Explicit-state model checkers such as JPF and SPIN will always be faced with the state-explosion problem — it can be pushed further away but it will never go away — therefore it seems inevitable that unless we employ clever abstractions [1] we will need to focus on error-detection, in which case the development of clever heuristics to guide the model checker towards likely errors will be a fruitful area of research.

It is our view that when doing a heuristic search not only should one gear the heuristics towards the property to be checked, but also one should focus the selection of the heuristics to be used on the structure of the program being analyzed. We showed two such structure-dependent heuristics here: namely, one geared towards finding interleaving related problems, and one for analyzing highly nondeterministic programs. Furthermore, one should allow the capability for the user to specify heuristics, since the user's domain knowledge will be invaluable during model checking.

Many areas for future research exists within heuristic-based model checking, most notably how to select the best heuristic and heuristic parameters for a specific problem. This area has seen much attention in the AI community and we hope to leverage their results. We also believe a closer link between the property specific language and heuristics should exist: we envisage a property including certain heuristic guidelines, e.g. we might like to specify "DFS until full(queue) then show no-deadlock using branch-exploration". The next phase of the heuristic-based JPF development will therefore focus on learning when to use which heuristic as well as the development of a language for guiding the model checker during property checking.

References

1. Thomas Ball, Sriram K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *SPIN 2001*, pages 103–122, 2001. LNCS 2057.
2. Gerard Holzmann and Doron Peled. The State of SPIN. In *CAV '96*, LNCS, 1996.
3. Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Directed explicit model checking with HSF-Spin. In *SPIN 2001*, pages 57–79, 2001. LNCS 2057.
4. Stefan Edelkamp, Alberto Lluch Lafuente, and Stefan Leue. Trail-Directed Model Checking. In *Proceedings of the Workshop of Software Model Checking*, Electrical Notes in Theoretical Computer Science, Elsevier, July 2001.
5. W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs. In *IEEE International Conference on Automated Software Engineering (ASE)*, September 2000.