Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Heuristics for Model Checking Java Programs

Alex Groce[1], **Willem Visser**[2]

[1] School of Computer Science, Carnegie Mellon University, Pittsburgh, PA 15213-3891, USA, e-mail: agroce@cs.cmu.edu
[2] RIACS, NASA Ames Research Center, Moffett Field, CA 94035, USA, e-mail: wvisser@email.arc.nasa.gov

The date of receipt and acceptance will be inserted by the editor

Abstract. Model checking of software programs has two goals: one is the verification of correct software. The other is the discovery of errors in faulty software. Some techniques for dealing with the most crucial problem in model checking, the state space explosion problem, concentrate on the first of these goals. In this paper we present an array of heuristic model checking techniques for combating the state space explosion when searching for errors. Previous work on this topic has mostly focused on property-specific heuristics closely related to particular kinds of errors. We present structural heuristics that attempt to explore the structure (branching structure, thread inter-dependency structure, abstraction structure) of a program in a manner intended to expose errors efficiently. Experimental results show the utility of this class of heuristics. In contrast to these very general heuristics, we also present very lightweight techniques for introducing program-specific heuristic guidance.

1 Introduction

There has been recent interest in model checking software written in real programming languages [3,11,24,34,35,52]. The primary challenge in software model checking, as in all model checking, is the state space explosion problem: exploring all of the behaviors of a system is, to say the least, difficult when the number of behaviors is exponential in the possible inputs, contents of data structures, or number of threads in a program. A vast array of techniques have been applied to this problem [9], first in hardware verification, and now, increasingly, in software verification [3,11,31]. Many of these techniques require considerable non-automatic work by experts or do not apply as well to software as to hardware. Most of these techniques are aimed at reducing the size of the total state space that must be explored, or representing it symbolically so as to reduce the memory and time needed for the exploration.

Abstraction techniques (and specifically predicate abstraction [26]) have proven to be useful for software model checking [3,31]. However, applying such abstractions is expensive if the number of predicates required becomes large and determining whether an abstract behavior is also possible in the concrete program can be undecidable. These techniques have therefore been used mostly to show properties dependent on the control-flow of a program rather than to analyze systems where the properties depend on data—e.g. show reachability of a statement or that a specific sequence of API calls are possible, rather than show that a real-time scheduler will always allocate each thread its requested time.

An alternative approach is to concentrate not on verifying the correctness of programs but on dealing with the state space explosion when attempting to find errors. Rather than reducing the overall size of the state space, we can attempt to find a counterexample before the state explosion exhausts memory. Therefore producing a counterexample can be seen as searching through the state space of a system for a specific (error) behavior. Rather than *blindly* searching through the state-space, as is common for traditional model checking, we can then focus on using heuristics to guide the search. Heuristic model checking therefore aims at generating counterexamples by searching the *bug-containing* part of the state space first. Obviously we do not know, in general, what part of a program's state space is going to contain an error, or even if there is an error present.

A separate motivation for heuristic search in bug-finding is that although one of the strongest advantages of model checking is the generation of counterexamples when verification fails, traditional depth-first search algorithms tend to return very long counterexamples; heuristic search, when it succeeds, almost always produces much more succinct counterexamples.

The use of heuristics in model checking has so far mostly focused on using the property to be checked as a measure for guiding the search [14,25,32,40,54]. Unfortunately, unlike in more traditional optimization problems in which heuristics are more commonly used, it is not always possible to know during model checking how *close* one is to a property being violated. Measurements of distance to assertion statements, possible uncaught exceptions, or deadlocks require a concentration on a particular error. In a large program with many possible errors, this results in either a search for each possible error or a heuristic that may be hopelessly conflicted (for example, if every thread contains an assertion, the heuristic will be trying to move forward on each thread). In this paper we propose also using the structure of programs to develop heuristics to guide the search. In particular we show how structural coverage and thread-interdependence can be used as heuristics for model checking. Furthermore, to illustrate that heuristics can also be used during abstraction-based model checking, we propose a heuristic that will reduce false positive results (errors that are possible in the abstract program, but not possible in the concrete), by always first searching for errors in the portion of the state space where no infeasible behaviors exist.

We believe one of the most interesting aspects of heuristic model checking is to learn from previous analyses which heuristic is best-suited to discovering an error in a program. To illustrate the vision, we show how a heuristic that favors the execution of a subset of threads in a program can be calibrated by selecting threads that formed part of race-violation (discovered during a previous model checking run)—the reasoning being that a race violation can lead to something more severe (which it did, since a deadlock was thus discovered).

Finally, we believe that the tester (or developer) using the model checker will in all likelihood know more about the overall structure of the program and in which parts there might be errors lurking than can be discovered automatically. Therefore, it is important to allow for user-defined heuristics. We propose an approach where the user can define a new heuristic function, or take a more light-weight approach and simply annotate the program with statements that the model checker can use during analysis to indicate which parts of the state space are most *interesting* to explore.

The contributions of the paper are:

- A suggested combination of property-based, structural and user-defined heuristics.
- Description of a set of structural heuristics that exploit structural coverage, concurrency structures, and structures induced by applying abstractions before model checking.
- Experimental results for the new heuristics within the context of error-detection with Java PathFinder (JPF) [52].

The paper is organized as follows. Section 2 describes heuristic model checking and presents the basic algorithms involved. Section 3 presents the Java PathFinder model checker and the implementation of heuristic search. Structural heuristics are defined and described in detail in Section 4, which also includes experimental results. Section 5 presents user-guided heuristics and heuristic annotations. The now considerable body of literature on heuristic model checking is described in section 6. We present our conclusions and consider future work in a final section.

2 Heuristic Model Checking

In *heuristic* or *directed* model checking, a state space is explored in an order dependent on an evaluation function for states. This function (the heuristic) is usually intended to guide the model checker more quickly to an error state. Any resulting counterexamples will often be shorter than ones produced by the depth-first search based algorithms traditionally used in explicit-state model checkers. Heuristic model checking is a growing field; we discuss the large body of related work in Section 6.

2.1 Search Algorithms

A number of different search algorithms can be combined with heuristics. All share a common structure: a fitness f is computed for each state generated by the model checker, and then the values for f are used to determine which states are explored next. The searches all terminate if a goal is reached (for our purposes, if a property violation is detected). f's value will be derived from a heuristic function h evaluating the state. Because many of the heuristic functions we use take into account the path by which a state was reached or other search-level information, the primary distinction between f and h in this paper is that f is used to introduce search-strategy specific modifications to heuristics that are always computed in the same fashion for any of these algorithms.

The simplest of heuristic search algorithms is a best-first search, which uses the heuristic function h to compute a fitness f in a greedy fashion (Figure 1).

The A^* algorithm [29] is similar, except that like Dijkstra's shortest paths algorithm, it adds the length of the path to S' to f (f = $h(S')$ + $\texttt{path-length}(S')$ rather than f = $h(S')$). When the heuristic function h is *admissible*, that is, when $h(S')$ is guaranteed to be less than or equal to the length of the shortest path from S' to a goal state, A^* is guaranteed to find an optimal solution (for purposes of model checking, the shortest coun-

```
priority queue Q = {initial state}
while (Q not empty)
    S = state in Q with best f
    remove S from Q
    for each successor state S' of S
    if S' not already visited
        if S' is the goal
                  terminate
            f = h(S')
        store (S', f) in Q
```

Fig. 1. Algorithm for *best-first* search.

```
queue Q = {initial state}
while (Q not empty)
    while (Q not empty)
        priority queue Q' = ∅
        remove S from Q
        for each successor state S' of S
        if S' not already visited
            if S' is the goal
                      terminate
                f = h(S')
            store (S', f) in Q'
    remove all but k best elements from Q'
    Q = Q'
```

Fig. 2. Algorithm for *beam* search.

terexample). A^* is a compromise between the guaranteed optimality of breadth-first search and the efficiency in returning a solution of best-first search.

Beam-search proceeds even more like a breadth-first search, but uses the heuristic function to discard all but the k best candidate states at each depth (Figure 2). Beam-search, therefore, can only be used to discover errors—termination of a beam search without discovering an error does not imply correctness of the system.

The queue-limiting technique used in beam-search may also be applied to a best-first or A^* search by removing the worst state from Q (without expanding its children) whenever inserting S' results in Q containing more than k states. This, again, introduces an incompleteness into the model checking run: termination without reported errors does not indicate that no errors exist in the state space. However, given that the advantage of heuristic search is its ability to quickly discover fairly short counterexamples, in practice queue-limiting is a very effective bug-finding tactic.

The experimental results in Section 4 show the varying utility of the different search strategies. Because none of the heuristics we examine are admissible, A^* lacks a theoretical optimality, and is generally less efficient than best-first search. The heuristic value is sometimes much larger than the path length, in which case A^* behaves much like a best-first search.

As far as we are aware, combining a best-first search with limitations on the size of the queue for storing states pending is not discussed or given a name in the literature of heuristic search. A best-first search with queue limiting can find very deep solutions that might be difficult for a beam-search to reach unless the queue limit k is very small. In practice, picking a k for either type of search is done by a hand approximation of iterative widening [50].

Introduction of queue-limiting to heuristic search for model checking raises the possibility of using other incomplete methods when the focus of model checking is on discovery of errors rather than on verification. As an example, partial order reduction techniques usually require a cycle check that may be expensive or overconservative in the context of heuristic search [15]. However, once queue-limiting is considered, it is natural to experiment with applying a partial order reduction without a cycle check. The general approach remains one of model checking rather than testing because storing of states already visited is crucial to obtaining good results in our experience, with one notable exception (see the discussion in sections 4.1.2 and 4.2.1).

3 Java PathFinder

Java PathFinder (JPF) is an explicit state on-the-fly model checker that takes compiled Java programs (i.e. bytecode class-files) and analyzes all paths through the program for deadlock, assertion violations and linear time temporal logic (LTL) properties [52]. JPF is unique in that it is built on a custom-made Java Virtual Machine (JVM) and therefore does not require any translation to an existing model checker's input notation. The *dSPIN* model checker [35] that extends SPIN [33] to handle dynamic memory allocation and functions is the most closely related system to the JPF model checker.

Java does not support nondeterminism, but in a model checking context it is often important to analyze the behavior of a program in an aggressive environment where all possible actions, in any order, must be considered. For this reason, methods in a special `Verify` class allow programs to express nondeterminism. For example, `Verify.random(2)` will nondeterministically return a value in the range 0–2, inclusive, which the model checker can then trap during execution and evaluate with all possible values.

An important feature of the model checker is the flexibility in choosing the granularity of a *transition* between states during the analysis of the bytecode. Since the model checker executes bytecode instructions, the most fine-grained analysis supported is at the level of individual bytecodes. Unfortunately, for large programs the bytecode-level analysis does not scale well, and therefore the default mode is to analyze the code on a line-by-line basis. JPF also supports *atomic* constructs (denoted by `Verify.beginAtomic()` and `Verify.endAtomic()` calls) that the model checker can trap to allow larger code fragments to be grouped into a single *transition*.

The model checker consists of two basic components:

State Generator - This includes the JVM, information about scheduling, and the state storage facilities required to keep track of what has been executed and which states have been visited. The default exploration in JPF is to do a depth-first generation of the state space with an option to limit the search to a maximum depth. By changing the scheduling information, one can change the way the state space is generated - by default a stack is used to record the states to be expanded next, hence the default DFS search.

Analysis Algorithms - This includes the algorithms for checking for deadlocks, assertion violations and violation of LTL properties. These algorithms work by instructing the state generation component to generate new states, backtrack from old states, and can check on the state of the JVM by doing API calls (e.g. to check when a deadlock has been reached).

The heuristics in JPF are implemented in the *State Generator* component, since many of the heuristics require information from the JVM and a natural way to do the implementation is to adapt the scheduling of which state to explore next (e.g. in the trivial case, for a breath-first search one changes the stack to a queue). Best-first (also used for A^*) and beam-search are straightforward implementations of the algorithms listed in Section 2.1, using priority queues within the scheduler. The heuristic search capabilities are currently limited to deadlock and assertion violation checks—none of the heuristic search algorithms are particularly suited to cycle detection, which is an important part of checking LTL properties. In addition, the limited experimental data on improving cycles in counterexamples for liveness properties is not encouraging [16].

Heuristic search in JPF also provides a number of additional features, including:

- users can introduce their own heuristics (interfacing with the JVM through a well-defined API to access program variables etc.)
- the sum of two heuristics can be used
- the order of analysis of states with the same heuristic value can be altered
- the number of elements in the priority queue can be limited
- the search depth can be limited
- dynamic annotations in the source code can cause the model checker to increase or decrease heuristic values or even remove parts of the search space

4 Structural Heuristics

Heuristics can be used in symbolic model checking to reduce the bottlenecks of image computation, without necessarily attempting to zero in on errors; Bloem, Ravi and Somenzi thus draw a distinction between *property-dependent* and *system-dependent* heuristics [5]. They note that only property-dependent heuristics can be applied to explicit-state model checking, in the sense that exploring the state space in a different order will not remove bottlenecks in the event that the entire space must be explored. We suggested a further classification of property-dependent heuristics into *property-specific heuristics* that rely on features of a particular property (queue sizes or blocking statements for deadlock, distance in control or data flow to false valuations for assertions) and *structural heuristics* that attempt to explore the structure of a program in a way conducive to finding more general errors [28]. The heuristic used in FLAVERS would be an example of the latter [10].

Previous work on model checking using heuristics largely concentrates on property-specific heuristics [14, 25,32,40,54]. Common heuristics include measuring the lengths of queues, giving preference to blocking operations [14,40], and using a Hamming distance to a goal state [16,54]. Heuristics tailored to match a property or derived statically from a combination of the source code and the property (such as distance to assertions or a search for overflow of a particular buffer) are certainly useful. However, when a model checker is applied to a large concurrent program with many assertions and the potential for deadlocks and uncaught exceptions, it is unclear how to pick a property-specific heuristic. Rather than looking for a specific error, it may be best to try to explore the structure of the program systematically, looking for any kind of error. As we note below, this is the motivation behind coverage metrics in testing.

We consider the following heuristics to be *structural heuristics* because they explore some structural aspect of the program (branching, thread-interdependence, etc.) independent of any specific property.

4.1 Code Coverage Heuristics

The code coverage achieved during testing is a measure of the adequacy of the testing—or, in other words, of the quality of the set of test cases. Although it does not directly address the correctness of the code under test, having achieved high code coverage during testing without discovering any errors does inspire more confidence that the code is correct. A case in point is the avionics industry where software can only be certified for flight if 100% structural coverage, specifically modified condition/decision coverage (MC/DC), is achieved during testing [47].

In the testing literature there are a vast number of structural code coverage criteria, from simply covering all statements in the program to covering all possible execution paths. Here we will focus on branch coverage, which requires that at every branching point in the program all possible branches be taken at least once.

In many industries 100% branch coverage is considered a minimum requirement for test adequacy [4]. On the face of it, one might wonder why coverage during model checking is of any value, since model checkers typically cover all of the state space of the system under analysis, hence by definition covering all the structure of the code. However, when model checking Java programs the programs are often infinite-state, or have a very large finite state space, which the model checker cannot cover due to resource limitations (typically memory). Calculating coverage therefore serves the same purpose as during testing: it shows the adequacy of the (partial) model checking run.

As with test coverage tools, calculating branch coverage during model checking only requires us to keep track of whether at each structural branching point all options were taken. Since JPF executes bytecode statements, this means simple extensions need to be introduced whenever `IF*` (related to any if-statement in the code) and `TABLESWITCH` (related to case-statements) are executed to keep track of the choices made. However, unlike with simple branch coverage, we also keep track of how many times each branch was taken, rather than just whether it was taken or not, and consider coverage separately for each thread created during the execution of the program. The first benefit of this feature is that the model checker can now produce detailed coverage information when it exhausts memory without finding a counterexample or searching the entire state space. Additionally, if coverage metrics *are* a useful measurement of a set of test cases, it seems plausible that using coverage as a heuristic to prioritize the exploration of the state space might be useful.

One approach to using coverage metrics in a heuristic would be to simply use the percentage of branches covered (on a per-thread or global basis) as the heuristic value (we refer to this as the %-coverage heuristic). However, this approach does not work well in practice (see Section 4.1.2). Instead, a slightly more complex heuristic proves far more useful:

1. States covering a previously untaken branch receive the best heuristic value.
2. States that are reached by not 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 (worse scores are assigned to more frequently taken branches).

The motivation behind the *branch counting* heuristic is to make use of the branching structure of a program while avoiding some of the pitfalls of the more direct heuristic.

The %-coverage heuristic is likely to fall into local minima, exploring paths that cover a large number of branches but do not in the future increase coverage. The branch counting heuristic behaves in an essentially breadth-first manner unless a path is actually increasing coverage. By default, JPF explores states with the same heuristic value in a FIFO manner, resulting in a breadth-first exploration of a program with no branch choices. However, if there are branch choices, the exploration will proceed in a manner that is not strictly breadth-first. Even after the coverage ceases to increase, the frontier is much deeper along paths which have previously increased coverage, so the search still advances exploration of structurally interesting paths over uninteresting paths.

```
public static void main (String [] args) {
  int x = Verify.random (2);
  int y = Verify.random (2);
  for (int i = 0; i < x; i++) {
    System.out.println
      ("x,y,i:" + x + "," + y + "," + i);
  }
  for (int j = 0; j < y; j++) {
    System.out.println
      ("x,y,j:" + x + "," + y + "," + j);
  }
}
```

Fig. 3. Example program for the branch counting heuristic.

The improved heuristic delays exploration of repetitive portions of the state space (those that take the same branches repeatedly). Choosing untaken branches (Rule 1) obviously cannot lead to repetitive exploration of a part of a system's control flow. While there is no guarantee of novelty in choosing transitions not involving branches (Rule 2), any repetition of these transitions is presumably guarded by a branch. Thus if a nondeterminisic choice determines how many times to execute a loop, for instance, it will delay exploring through multiple iterations of the loop along certain paths until it has searched further along paths that skip the loop or execute it only once. The heuristic thus achieves deeper coverage of the structure and examines possible behaviors after termination of the loop. If the paths beyond the loop continue to be free of branches or involve previously uncovered branches, exploration will continue; however, if one of these paths leads to a loop, exploration will return to explore further iterations of the first loop before executing the latter loop more than once, due to the third rule.

In order to clarify the consequences of this ordering, we present a small example program (Figure 3) and show the results of using a few different search strategies to explore it (Table 1). The first column for each search is the output (in order). Beside each output is the count for the branches at that point in execution: the first two numbers are the true and false branches for the branch on `i < x` and the second and third numbers are the true and false branches for `j < y`. DFS immediately diverges from the other two strategies: the model checker initially

DFS		BFS		branchcount	
j:0,1,0	0/2/1/1	i:1,0,0	6/3/2/1	i:1,0,0	1/3/0/1
j:0,2,0	0/3/2/2	i:1,1,0	6/3/2/1	j:0,1,0	3/3/1/1
j:0,2,1	0/3/3/2	i:1,2,0	6/3/2/1	i:1,1,0	3/3/1/1
i:1,0,0	1/3/3/3	i:2,0,0	6/3/2/1	j:0,2,0	6/3/2/1
i:1,1,0	2/4/3/4	i:2,1,0	6/3/2/1	i:1,2,0	6/3/2/1
j:1,1,0	2/5/4/4	i:2,2,0	6/3/2/1	i:2,0,0	6/3/2/1
i:1,2,0	3/5/4/5	j:0,1,0	6/3/2/1	i:2,1,0	6/3/2/1
j:1,2,0	3/6/5/5	j:0,2,0	6/3/2/1	i:2,2,0	6/3/2/1
j:1,2,1	3/6/6/5	i:2,0,1	9/6/5/3	j:1,1,0	6/5/3/3
i:2,0,0	4/6/6/6	i:2,1,1	9/6/5/3	j:0,2,1	6/5/4/3
i:2,0,1	5/6/6/6	i:2,2,1	9/6/5/3	j:1,2,0	6/6/5/3
i:2,1,0	6/7/6/7	j:0,2,1	9/6/5/3	i:2,0,1	7/6/5/3
i:2,1,1	7/7/6/7	j:1,1,0	9/6/5/3	i:2,1,1	8/6/5/3
j:2,1,0	7/8/7/7	j:1,2,0	9/6/5/3	i:2,2,1	9/6/5/3
i:2,2,0	8/8/7/8	j:1,2,1	9/9/8/6	j:1,2,1	9/6/6/5
i:2,2,1	9/8/7/8	j:2,1,0	9/9/8/6	j:2,1,0	9/8/7/6
j:2,2,0	9/9/8/8	j:2,2,0	9/9/8/6	j:2,2,0	9/9/8/6
j:2,2,1	9/9/9/8	j:2,2,1	9/9/9/8	j:2,2,1	9/9/9/8

Table 1. Order of exploration for code in Figure 3.

chooses the pair (0, 0) for (x, y) (which causes nothing to be printed, as neither for-loop can be entered). Backtracking then causes the model checker to choose a second value for y, resulting in the pair (0, 1). As this is a depth-first search, the first loop is therefore skipped again, but the second (j) loop is entered, printing (0, 1, j = 0). BFS and branch counting, on the other hand, both first show the print statement that can be reached in the fewest steps, in the first (i) loop. However, they diverge immediately afterwards. The same print statement can also be reached at the same search depth with y's value being 1. As the branch counting heuristic always prefers a path with an unexplored branch, it first shows the execution where the first loop is never executed but the second is executed once. At this point, each branch in the loops has been explored by branch counting (as with DFS, the case where both branches are not taken is invisible).

Additionally, it is important to note that while BFS and branchcount both display `x,y,i:1,0,0` first, the branch coverage counts are quite different. In the BFS, all of the possible combinations for x and y are generated and the first `for` loop is executed with each possible combination. There are nine possible choices for x and y, 6 of which cause the true branch to be taken and 3 of which cause the false branch to be taken. These have *all* been executed before the first print: thus the true branch has been covered 6 times and the false branch 3 times. In contrast, for branchcount, the print statement is covered when the true branch has only been executed once. BFS, therefore, results in a very static set of coverages, with only 3 total changes to the coverage counts for each branch. For branchcount, the coverage numbers change a total of 12 times. While DFS changes coverage between every two printings, it does so in a non-methodical manner that is weighted towards the false branches—it increases the counts, but makes no directed effort to increase *coverage*. Thus while both BFS and branchcount have covered all branches by the second printing, DFS does not do so until the fourth printing. The behavior of the branchcount heuristic can be seen as a mixture of the behaviors of BFS and DFS that is sensitive to the branching structure of the program.

In addition to avoiding local minima, this heuristic has the advantage of being more sensitive to data values than the coverage measures traditionally used in testing. Because the heuristic *counts* the executions of each branch, it is influenced by data values that determine how many times a for-loop is executed in a manner beyond the simple 0-1 sensitivity of all-or-nothing coverage.

4.1.1 Variations on the Branch Counting Heuristic.

A number of options can modify the basic strategy:

- Counts may be taken globally (over the entire state space explored) or only for the path by which a particular state is reached. This allows us to examine either combinations of choices along each path or to try to maximize branch choices over the entire search when the ordering along paths is less relevant. In principle, the path-based approach should be useful when taking certain branches in a particular combination in an execution is responsible for errors. Global counts will be more useful when simply exercising all of the branches is a better way to find an error. An instance of the latter would be a program in which one large nondeterministic choice at the beginning results in different classes of shallow executions, one of which leads to an error state.
- The branch count may be allowed to persist—if a state is reached without covering any branches, the last branch count on the path by which that state was reached may be used instead of giving the state the second best heuristic value. This allows us to increase the tendency to explore paths that have improved coverage without being quite as prone to falling into local minima as the %-coverage heuristic.
- The counts over a path can be summed to reduce the search's sensitivity to individual branch choices.
- These various methods can also be applied to counts taken on executions of each individual bytecode instruction, rather than only of branches. This is equivalent to the idea of *statement coverage* in traditional testing.

The practical effect of this class of heuristic is to increase exploration of portions of the state space in which nondeterministic choices or thread interleavings have resulted in the possibility of previously unexplored or less-explored branches being taken.

In practice, these variations behaved very much like the basic branch counting heuristic in our experimental results. For persistent and summing counts, in fact, the results were identical to the standard search in almost all cases, and were thus omitted from the experimental results. In theory, all of the variations can produce significantly different results on real programs, but in practice only global vs. path had any observable impact.

Note also that the branch counting heuristics can be used in dynamic test case generation [39] by using the heuristic function to optimize the selection of test cases—for example, by only picking cases in which the coverage increases.

4.1.2 Experimental Results

We will refer to a number of heuristics (Table 2) and search strategies (Table 3) when presenting experimental results. In addition to these basic heuristics, we indicate whether a heuristic is measured over paths or all states by appending *(path)* or *(global)* when that is an option.

The DEOS real-time operating system developed by Honeywell enables Integrated Modular Avionics (IMA) and is currently used within certain small business aircraft to schedule time-critical software tasks. During its development a routine code inspection led to the uncovering of a subtle error in the time-partitioning that could allow tasks to be starved of CPU time - a sequence of unanticipated API calls made near time-period boundaries would trigger the error. Interestingly, although avionics software needs to be tested to a very high degree (100% MC/DC coverage) to be certified for flight, this error was not uncovered during testing. Model checking was used to rediscover this error, by using a translation to PROMELA (the input language of the SPIN model checker) [44]. Later a Java translation of the original C++ code was used to detect the error. Both versions use an abstraction to find the error (see the discussion in section 4.4). The results (Table 4) are from a version of the Java code that does not abstract away an infinite-state counter—a more straightforward translation of the original C++ code into Java.

The %-coverage heuristic does indeed appear to easily become trapped in local minima, and, as it is not admissible, using an A^* search will not necessarily help. For comparison to results not using heuristics, here and below we also give results for breadth-first search (BFS), depth-first search (DFS) and depth-first searches limited to a certain maximum depth. For essentially infinite state systems (such as this version of DEOS), limiting the depth is the only practical way to use DFS, but as can be seen, finding the proper depth can be difficult—and large depths may result in extremely long counterexamples. Using a purely random heuristic does, in fact, find a counterexample for DEOS—however, the counterexample is considerably longer and takes more time and memory to produce than with the coverage heuristics.

We also applied the successful heuristics to the DEOS system with the storing of visited states turned off (performing testing or simulation rather than model checking, essentially). Without state storage, these heuristics failed to find a counterexample before exhausting memory—the queue of states to explore becomes too large and exhausts the memory.

4.2 Thread Interleaving Heuristics

A different kind of structural heuristic is based on maximizing thread interleavings. Testing, in which generally the scheduler cannot be controlled directly, often misses subtle race conditions or deadlocks because they rely on unlikely thread scheduling. One way to expose concurrency errors is to reward "demonic" scheduling by assigning better heuristic values to states reached by paths involving more switching of threads. In this case, the structure we attempt to explore is the dependency of the threads on precise ordering. If a non-locked variable is accessed in a thread, for instance, and another thread can also access that variable (leading to a race condition that can result in a deadlock or assertion violation), that path will be preferred to one in which the accessing thread continues onwards, perhaps escaping the effects of the race condition by reading the just-altered value. This heuristic is calculated by keeping a (possibly limited in size) history of the threads scheduled on each path:

- At each step of execution append the thread just executed to a thread history.
- Pass through this history, making the heuristic value that will be returned worse each time the thread just executed appears in the history by a value proportional to:
  1. how far back in the history that execution is and
  2. the current number of live threads

Figure 4 presents a small sample program and Table 5 shows how the interleaving heuristic affects the order of exploration of its state space by the model checker. As with branch counting, DFS is immediately distinguishable from the other heuristics, as the model checker executes Thread #1 until this is no longer possible (after which we observe backtracking behavior). BFS and the interleaving heuristic both behave very similarly at first; as with the branch counting heuristic, the basic approach is to imitate a breadth-first exploration until information is available to modify this behavior. Figure 5 shows the beginning of the possible interleavings of the print statements for the code. There are only two possible print statements at the earliest depth at which a print statement can be encountered. States with the same heuristic value are ordered by the creation order of the threads. At the second depth at which print statements can be encountered, there are four choices, but the thread history

Heuristic	Definition
branchcount	The basic branch counting heuristic. Has multiple variations, such as path or global coverage.
bytecode	Computed in the same manner as the branchcount heuristic, but applied to all bytecode instructions; also comes in path and global variations.
%-coverage	Measures the percentage of branches covered. States with higher coverage receive better values.
most-blocked	Measures the number of blocked threads. More blocked threads results in better values.
interleaving	Measures the amount of interleaving of threads on paths. See Section 4.2.
prefer-threads	Uses heuristic value to prefer execution of a given set of threads.
choose-free	Uses heuristic value to avoid abstraction-introduced nondeterminism.
random	Uses a randomly assigned heuristic value. Results shown are best of a series of runs.

Table 2. Heuristics.

Search strategy	Definition
BFS	A breadth-first search.
DFS	A depth-first search. (depth n) indicates that stack depth is limited to n.
best	Best-first search, (with possible queue limit k)
A^*	An A^* search (with possible queue limit k)
beam	Beam search (with a given k)

Table 3. Search strategies.

Search	Heuristic	Time(s)	Mem(MB)	States	Length	Max Depth
best	branchcount (path)	60	92	2,701	136	139
A^*	branchcount (path)	59	90	2,712	136	139
best	branchcount (global)	60	91	2,701	136	139
A^*	branchcount (global)	59	92	2,712	136	139
best	bytecode (path)	-	FAILS	9,032	-	168
A^*	bytecode (path)	-	FAILS	10,073	-	139
best	bytecode (global)	62	88	2,195	136	137
A^*	bytecode (global)	63	94	2,383	136	137
best	%-coverage (path)	-	FAILS	20,215	-	334
A^*	%-coverage (path)	-	FAILS	18,141	-	134
best	%-coverage (global)	-	FAILS	20,213	-	334
best	random	162	240	8,057	334	360
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
Results with state storage turned off						
best	branchcount (path)	-	FAILS	15,964	-	125
A^*	branchcount (path)	-	FAILS	15,962	-	125
best	branchcount (global)	-	FAILS	15,964	-	125
A^*	branchcount (global)	-	FAILS	15,962	-	125

Table 4. Experimental results for the DEOS system.

All results obtained on a 1.4 GHz Athlon with JPF limited to 512Mb. **Time(s)** is in seconds and **Mem(MB)** is in megabytes. **FAILS** indicates failure due to running out of memory. The **Length** column reports the length of the counterexample (if one is found). The **Max Depth** column reports the length of the longest path explored (the maximum stack depth in the depth-first case).

```
class MyThread extends Thread {
  public static int s = 0;
  private int tid;
  MyThread (int i) {
    tid = i;
  }
  public void run(){
    System.out.println ("Thread #" + tid + ", A");
    System.out.println ("Thread #" + tid + ", B");
    System.out.println ("Thread #" + tid + ", C");
  }
}
class IntExample {
  public static void main (String [] args) {
    Verify.beginAtomic ();
    MyThread Thread1 = new MyThread(1);
    MyThread Thread2 = new MyThread(2);
    Thread1.start (); Thread2.start ();
    Verify.endAtomic ();
  }
}
```

Fig. 4. Example program for interleaving heuristic.

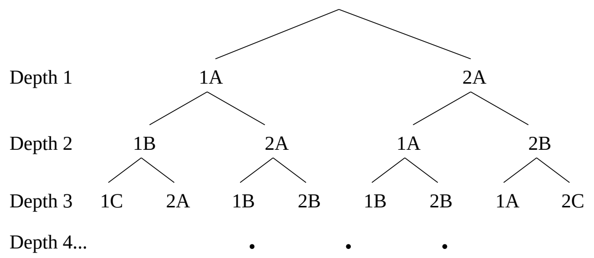

Fig. 5. Thread interleaving for code in Figure 4.

DFS	BFS	interleaving
Thread #1, A	Thread #1, A	Thread #1, A
Thread #1, B	Thread #2, A	Thread #2, A
Thread #1, C	Thread #1, B	Thread #1, B
Thread #2, A	Thread #2, A	Thread #2, A
Thread #2, B	Thread #1, A	Thread #1, A
Thread #2, C	Thread #2, B	Thread #2, B
Thread #2, A	Thread #1, C	Thread #1, B
Thread #1, C	Thread #2, A	Thread #2, B
Thread #2, B	Thread #1, B	Thread #1, B
Thread #1, C	Thread #2, B	Thread #2, B
Thread #2, C	Thread #1, B	Thread #1, C
Thread #1, C	Thread #2, B	Thread #2, A
Thread #2, A	Thread #1, A	Thread #2, A
Thread #1, B	Thread #2, C	Thread #2, B
Thread #2, B	Thread #2, A	Thread #2, C
Thread #1, B	Thread #1, C	Thread #1, A
Thread #2, C	Thread #2, B	Thread #2, C
Thread #1, B	Thread #1, B	Thread #1, A
Thread #2, A	Thread #2, C	Thread #1, B
Thread #1, A	Thread #1, C	Thread #1, C
Thread #1, B	Thread #2, B	Thread #1, C
Thread #1, C	Thread #1, B	Thread #2, B
Thread #2, B	Thread #2, C	Thread #1, B
Thread #2, C	Thread #1, A	Thread #2, C
Thread #2, B	Thread #2, B	Thread #1, B
Thread #1, C	Thread #1, C	Thread #2, C
Thread #2, C	Thread #2, C	Thread #1, B
Thread #1, C	Thread #1, B	Thread #1, C
Thread #2, B	Thread #2, B	Thread #1, C
Thread #1, B	Thread #1, C	Thread #2, B
Thread #2, C	Thread #2, C	Thread #2, B
Thread #1, B	Thread #1, B	Thread #2, C
Thread #2, B	Thread #2, C	Thread #1, C
Thread #1, A	Thread #1, C	Thread #2, C
Thread #2, C	Thread #2, C	Thread #1, C
Thread #1, A	Thread #1, C	Thread #2, C

Table 5. Order of exploration for code in Figure 4.

is too small to activate the interleaving heuristic. It is at the third depth (indicated by the line in Table 5) that we see divergence between the BFS and the interleaving heuristic. 1C, the first print statement at this depth in the BFS, results from a very non-interleaved execution sequence in which thread #1 is chosen three times in a row. 1B, the interleaving choice, results from executing thread #1, then thread #2, then thread #1 again. 2A, the next BFS choice, must result from a path which begins by executing thread #1 twice in a row, while the interleaving heuristic causes 2B to appear first, as it can be reached by executing thread #2, then thread #1, and then thread #2 again. After this, the execution orders grow more divergent as more thread history is accumulated. The interleaving heuristic not only rearranges the order within a particular depth, it abandons breadth first search completely. A 2A execution from depth 4 appears before the 1A execution for depth 3.

4.2.1 Experimental Results

During May 1999 the Deep-Space 1 spacecraft ran a set of experiments whereby the spacecraft was under the control of an AI-based system called the Remote Agent. Unfortunately, during one of these experiments the software went into a deadlock state, and had to be restarted from Earth. The cause of the error at the time was unknown, but after some study, in which the most likely components to have caused the error were identified, the error was found by applying model checking to a Java version of the code—the error was due to a missing critical section causing a race violation to occur under certain thread interleavings introducing a deadlock [30]. The results (Table 6) use a version of the code that is faithful to the original system, as it also includes parts of the system not involved in the deadlock.

Experiments indicate that while A^* and beam-search can certainly perform well at times, they generally do not perform as well as best-first search. The heuristics investigated are not admissible, so the optimality advantages of A^* do not come into play. In general, both appear to require more judicious choice of queue-limits than is necessary with best-first search, at least in this example.

Search	Heuristic	Time(s)	Mem(MB)	States	Length	Max Depth
best ($k = 40$)	branchcount (path)	-	FAILS	1,765,009	-	12,092
best ($k = 160$)	branchcount (path)	-	FAILS	1,506,725	-	5,885
best ($k = 1000$)	branchcount (path)	132	290	845,263	136	136
best ($k = 40$)	branchcount (global)	-	FAILS	1,758,416	-	12,077
best ($k = 160$)	branchcount (global)	-	FAILS	1,483,827	-	1,409
best ($k = 1000$)	branchcount (global)	-	FAILS	1,509,810	-	327
best	random	-	FAILS	55,940	-	472
BFS	-	-	FAILS	623,566	-	60
DFS	-	-	FAILS	267,357	-	267,357
DFS (depth 500)	-	43	54	116,071	500	500
DFS (depth 1000)	-	44	64	117,235	1000	1000
DFS (depth 4000)	-	47	72	122,513	4000	4000
best	interleaving	-	FAILS	378,068	-	81
best ($k = 5$)	interleaving	15	17	38,449	913	913
best ($k = 40$)	interleaving	116	184	431,752	869	869
best ($k = 160$)	interleaving	908	501	1,287,984	869	870
best ($k = 1000$)	interleaving	-	FAILS	745,788	-	177
A^*	interleaving	-	FAILS	369,166	-	81
A^* ($k = 5$)	interleaving	13	19	43,172	912	912
A^* ($k = 40$)	interleaving	77	129	306,285	865	867
A^* ($k = 160$)	interleaving	-	FAILS	1,309,561	-	789
A^* ($k = 1000$)	interleaving	-	FAILS	1,836,675	-	273
beam ($k = 5$)	interleaving	14	16	35,514	927	927
beam ($k = 40$)	interleaving	91	113	238,945	924	924
beam ($k = 160$)	interleaving	386	418	1,025,595	898	898
beam ($k = 1000$)	interleaving	-	FAILS	1,604,940	-	365
best	most-blocked	7	33	7,537	158	169
best ($k = 5$)	most-blocked	-	FAILS	922,433	-	27,628
best ($k = 40$)	most-blocked	-	FAILS	913,946	-	4,923
best ($k = 160$)	most-blocked	-	FAILS	918,575	-	1,177
best ($k = 1000$)	most-blocked	6	10	7,537	158	169
A^*	most-blocked	-	FAILS	631,274	-	61
A^* ($k = 5$)	most-blocked	-	FAILS	935,796	-	16,189
A^* ($k = 40$)	most-blocked	-	FAILS	960,259	-	1,907
A^* ($k = 160$)	most-blocked	-	FAILS	989,513	-	555
A^* ($k = 1000$)	most-blocked	-	FAILS	1,138,920	-	165
best	prefer-threads	-	FAILS	548,157	-	61
best ($k = 5$)	prefer-threads	3	3	3,632	121	121
best ($k = 40$)	prefer-threads	6	12	23,754	121	121
best ($k = 160$)	prefer-threads	16	39	81,162	121	121
best ($k = 1000$)	prefer-threads	80	201	450,035	121	121

Table 6. Experimental results for the Remote Agent system.

Finally, for the dining philosophers (Table 7), we show that the interleaving heuristic can scale to quite large numbers of threads. While DFS fails to uncover counterexamples even for small problem sizes, the interleaving heuristic can produce short counterexamples for up to 64 threads. The most-blocked heuristic, designed to detect deadlocks, generally returns larger counterexamples (in the case of size 8 and queue limit 5, larger by a factor of over a thousand) after a longer time than the interleaving heuristic. Even more importantly, it does not scale well to larger numbers of threads. We only report, for each number of philosopher threads, the results for those searches that were successful in the next smaller version of the problem. Results not shown indicate that, in fact, failed searches do not tend to succeed for larger sizes.

The key difference in approach between using a property-specific heuristic and a structural heuristic can be seen in the dining philosophers example where we search for the well-known deadlock scenario. When increasing the number of philosophers high enough (for example to 16) it becomes impossible for an explicit-state model checker to try all the possible combinations of actions to get to the deadlock and heuristics (or luck) are required. A property-specific heuristic applicable here is to try and maximize the number of blocked threads (most-blocked heuristic from Table 2), since if all threads are blocked we have a deadlock in a Java program. Whereas a struc-

Search	Heuristic	Size	Time(s)	Mem(MB)	States	Length	Max Depth
best	branchcount (path)	8	-	FAILS	374,152	-	41
best	random	8	-	FAILS	218,500	-	86
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
best	most-blocked	8	-	FAILS	310,317	-	285
best ($k = 5$)	most-blocked	8	17,259	378	891,177	78,353	78,353
best ($k = 40$)	most-blocked	8	10	7	13,767	273	273
best ($k = 160$)	most-blocked	8	10	12	25,023	172	172
best ($k = 1000$)	most-blocked	8	46	59	123,640	254	278
best	interleaving	8	-	FAILS	487,942	-	16
best ($k = 5$)	interleaving	8	2	1	1,719	66	66
best ($k = 40$)	interleaving	8	5	5	16,569	66	66
best ($k = 160$)	interleaving	8	12	27	62,616	66	66
best ($k = 1000$)	interleaving	8	60	137	354,552	67	67
best ($k = 5$)	most-blocked	16	-	FAILS	802,526	-	36,443
best ($k = 40$)	most-blocked	16	38	69	101,576	1,008	1,008
best ($k = 160$)	most-blocked	16	-	FAILS	799,453	-	2,071
best ($k = 1000$)	most-blocked	16	-	FAILS	791,073	-	702
best ($k = 5$)	interleaving	16	4	5	6,703	129	129
best ($k = 40$)	interleaving	16	16	45	69,987	131	131
best ($k = 160$)	interleaving	16	60	207	290,637	131	132
best ($k = 1000$)	interleaving	16	-	FAILS	858,818	-	41
best ($k = 40$)	most-blocked	32	-	FAILS	463,414	-	2,251
best ($k = 5$)	interleaving	32	11	32	25,344	257	257
best ($k = 40$)	interleaving	32	-	FAILS	472,022	-	775
best ($k = 160$)	interleaving	32	-	FAILS	494,043	-	86
best ($k = 5$)	interleaving	64	59	206	101,196	514	514

Table 7. Experimental results for dining philosophers.

tural heuristic may be to observe that we are dealing here with a highly concurrent program—hence it may be argued that any error in it may well be related to an unexpected interleaving—hence we use the heuristic to favor increased interleaving during the search (interleaving heuristic from Table 2). Although the results are by no means conclusive, it is still worth noting that for this specific example the structural heuristic performs much better than the property-specific heuristic.

For the dining philosophers and Remote Agent example we also performed the experiment of turning off state storage. For the interleaving heuristic, results were essentially unchanged (minor variations in the length of counterexamples and number of states searched). We believe that this is because to return to a previously visited state in each case requires an action sequence that will not be given a good heuristic value by the interleaving heuristic (for example in the dining philosophers, alternating picking up and dropping of forks by the same threads). For the most-blocked heuristic, however, successful searches become unsuccessful—removal of state storage introduces the possibility of non-termination into the search. For example, the most-blocked heuristic without state storage may not even terminate, in some cases (imagine a scenario in which deadlock is impossible but a certain thread can acquire a lock, blocking all other threads—it is then forced to release the lock as the only thread that can execute, but the heuristic will then cause it to acquire the lock again, returning to the previous state).

Godefroid and Khurshid apply their genetic algorithm techniques to a very similar implementation of the dining philosophers (written in C rather than Java) [25]. They seed their genetic search randomly on a version with 17 running threads, reporting a 50% success rate and average search time of 177 seconds (on a slower machine than we used). The results suggest that the differences may be as much a result of the heuristics used (something like most-blocked vs. the interleaving heuristic) as the genetic search itself. Application of structural heuristics in different search frameworks is an interesting avenue for future study.

4.3 The Thread Preference Heuristic

The interleaving heuristic rewards context-switching; if a program has a large number of threads that are enabled, in an interleaving-guided search those threads will all be executed often. However, a few threads in a program may be suspected to harbor an error. Instead of rewarding context-switching (or trying to block threads that may be unrelated to the error), we can improve the heuristic value of transitions that involve execution of only these threads.

The Remote Agent example includes a scalable number of threads that are not involved in the actual error. If the size of this irrelevant part of the state space is increased, the various searches listed above perform much more poorly, and in many cases cannot find the error. The existence of the prefer-threads heuristic allows a new approach to hunting concurrency errors in JPF. JPF incorporates a version of the Eraser race detection algorithm [49]. First, the model checker is run in race detection mode with a small queue limit (300 worked well) and some heuristic (a BFS sufficed in our experiments). This search does not find the error, but reports a number of potential race conditions. Allowing the race detection to run for 3 minutes (using 389MB of memory) reveals that the Executive and Planner threads have unprotected write accesses to a field. The threads involved in the potential race conditions are then used to guide a thread-preference search with a similarly small queue, and a counterexample is quickly detected. This approach scaled to larger versions of the Remote Agent than other heuristics could handle (Table 8). The first block of results are for a version in which the irrelevant portion of the state space is doubled with respect to the version in Table 6, and the size is again doubled in each block (with only the searches succeeding for the last size shown).

This is a different flavor of structural heuristic than those presented previously. The branch counting and interleaving heuristics are not only not property-specific, but do not rely on specific knowledge of the program's behavior beyond what can be observed by the model checker (which branches are taken, which threads are enabled) during execution of the program. Preferring certain threads assumes knowledge about the behavior of the program; while it is not a property-specific heuristic, it focuses on a part of the system's structure guided by knowledge of what parts of the system are "interesting." However, our strategy with the Remote Agent demonstrates that this knowledge itself may be automatically extractable by the model checker. The experimental results show that such additional knowledge can, as would be expected, aid a guided search considerably.

4.4 The Choose-free Heuristic

Abstraction based on over-approximations of the system behavior is a popular technique for reducing the size of the state space of a system to allow more efficient model checking [8,12,26,53]. JPF supports two forms of over-approximation: predicate abstraction [53] and type-based abstractions (via the BANDERA tool) [12]. However, over-approximation is not well suited for error-detection, since the additional behaviors introduced by the abstraction can lead to spurious errors that are not present in the original. Eliminating spurious errors is an active area of research within the model checking community [3,7,31,43,48].

JPF uses a novel technique for the elimination of spurious errors called *choose-free* search [43]. This technique is based on the fact that all over-approximations introduce nondeterministic choices in the abstract program that were not present in the original. Therefore, a choose-free search first searches the part of the state space that doesn't contain any nondeterministic choices due to abstraction. If an error is found in this so-called choose-free portion of the state space then it is also an error in the original program. Although this technique may seem almost naive, it has been shown to work remarkably well in practice [12,43]. The first implementation of this technique was by *only* searching the choose-free state space, but the current implementation uses a heuristic that gives the best heuristic values to the states with the fewest nondeterministic choice statements enabled, i.e. allowing the choose-free state space to be searched *first* but continuing to the rest of the state space otherwise (this also allows choose-free to be combined with other heuristics).

The DEOS example can be abstracted by using both predicate abstraction [53] and type-based abstraction [12]. The predicate abstraction of DEOS is a precise abstraction, i.e. it does not introduce any new behaviors not present in the original, hence we focus here on the type-based abstraction—specifically we use a *Range* abstraction (allowing the values 0 and 1 to be concrete and all values 2 and above to be represented by one abstract value) to the appropriate variable [12]. When using the choose-free heuristic it is discovered that for this *Range* abstraction the heuristic search reports a choose-free error of length 26 in 20 seconds. The error path is shorter than in the experimental results reported earlier because those results are for a version of DEOS in which time is not abstracted (and thus arithmetic is not reduced to operations on the range-abstracted values).

These heuristics for finding feasible counterexamples during abstraction can be seen as an on-the-fly under-approximation of an over-approximation (from the abstraction) of the system behavior. The only other heuristic that we are aware of that falls into a similar category is the one for reducing infeasible execution sequences in the FLAVERS tool [10]. This heuristic differs from

Search	Heuristic	Time(s)	Mem(MB)	States	Length	Max Depth
Size = 10 (Table 6 is Size = 5 results)						
best ($k = 1000$)	branchcount (path)	-	FAILS	1,193,730	-	230
DFS (depth 500)	-	-	FAILS	599,431	-	500
DFS (depth 1000)	-	-	FAILS	598,487	-	1000
DFS (depth 4000)	-	-	FAILS	590,259	-	4000
best ($k = 5$)	interleaving	116	161	243,910	2,870	2,870
best ($k = 40$)	interleaving	-	FAILS	908,353	-	1,755
best ($k = 160$)	interleaving	-	FAILS	1,146,152	-	809
A^* ($k = 5$)	interleaving	112	158	241,999	2,867	2,867
A^* ($k = 40$)	interleaving	-	FAILS	892,071	-	1,764
beam ($k = 5$)	interleaving	116	151	209,370	2,888	2,888
beam ($k = 40$)	interleaving	-	FAILS	875,752	-	1,927
beam ($k = 160$)	interleaving	-	FAILS	1,066,711	-	902
best	most-blocked	25	186	23,528	269	280
best ($k = 1000$)	most-blocked	15	31	24,528	269	280
best ($k = 5$)	prefer-threads	4	8	11,137	201	201
best ($k = 40$)	prefer-threads	16	49	79,354	201	201
best ($k = 160$)	prefer-threads	51	169	290,932	201	201
best ($k = 1000$)	prefer-threads	-	FAILS	995,617	-	149
Size = 20						
best ($k = 5$)	interleaving	-	FAILS	639,748	-	5,321
A^* ($k = 5$)	interleaving	-	FAILS	635,067	-	5,031
beam ($k = 5$)	interleaving	-	FAILS	611,495	-	5,797
best	most-blocked	-	FAILS	41,991	-	497
best ($k = 1000$)	most-blocked	-	FAILS	402,007	-	524
best ($k = 5$)	prefer-threads	9	38	38,147	361	361
best ($k = 40$)	prefer-threads	55	272	286,554	361	361
best ($k = 160$)	prefer-threads	-	FAILS	680,990	-	279
Size = 40						
best ($k = 5$)	prefer-threads	38	212	140,167	681	681
best ($k = 40$)	prefer-threads	-	FAILS	472,708	-	445

Table 8. Larger versions of the Remote Agent.

those discussed previously in that it relies on the structure of the abstraction applied to a program rather than on the branching or thread-interaction structure of the program.

5 User-Guided Searches

5.1 User-Defined Heuristics

Traditionally, heuristics are often very problem-specific. Previous discussion throughout this paper has been of heuristics of general utility, but JPF allows for very specific heuristics as well. Users may write their own heuristics in Java. Consider a program with a class `Main` with a static field `buffer`, itself an object of a class with integer fields `current` and `capacity`. Figure 6 shows 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;
}
```

Fig. 6. Example of a user-defined heuristic.

Structural heuristics and property-specific heuristics of very general utility (such as the most-blocked heuristic) are provided as built-in features of the model checker, but the range of property-specific (or experimental structural heuristics) is so large that it is essential to allow users to also craft their own heuristics.

Method
`Verify.interesting (boolean b)`
If `b` **evaluates to true:** heuristic value for state in which the call appears is equal to the best possible heuristic value.
`Verify.boring (boolean b)`
If `b` **evaluates to true:** heuristic value for state in which the call appears is one step worse than the worst heuristic value previously computed in the model checking run.
`Verify.ignoreIf (boolean b)`
If `b` **evaluates to true:** the state in which the call appears is not explored by the model checker. This applies even to non-heuristic searches. Note that this potentially introduces an incompleteness into the search and can be dangerous if used unwisely.

Table 9. Special methods for program guided search.

```
public static void main (String [] args) {
  int x = Verify.random(10);
  int y = 0;
  Verify.interesting (x > 5);
  Verify.boring (x < 5);
  if (x > 5)
    y = 100;
  if (x < 5)
    y = 50;
  System.out.println ("y = " + y);
}
```

Fig. 7. Example program for program guided search.

Calls	Order
none	50, 100, 50, 100, 50, 100, 50, 100, 50, 100, 0
both	100, 50, 100, 100, 100, 100, 0, 50, 50, 50, 50

Table 10. Search ordering for example in Figure 7.

5.2 Program-Guided Search

A more lightweight approach than introducing new heuristics into the model checker itself is to introduce calls that are trapped by the model checker and used to modify the behavior of whatever heuristic is being used. JPF provides three methods for this purpose (Table 9).

These methods can be used to fine tune the behavior of the various heuristics provided by JPF in a dynamic fashion, based on values computed by the program being model checked at run time. The heuristic alteration from `Verify.interesting` and `Verify.boring` only applies to one state, but may have a significant effect on the search nonetheless. For example, if the branch counting heuristic is used and the successor to the interesting state covers a new branch, it (and possibly its successors) will be explored before the other states that would otherwise have had the same value as the interesting state.

As an example, consider the program in Figure 7. Model checking the program using the branch counting heuristic alone causes the model checker to alternate between executions outputting 50 and 100 because this keeps the counts on the branches even. Introducing calls to `Verify.interesting` and `Verify.boring` causes the first value printed to be 100, as the successors to the "boring" states are placed later in the queue. A single 50 then appears, as the branch counting heuristic's Rule 1 forces the first coverage of a branch to always have the best heuristic value. The other choices in which `(x < 5)` are all delayed until after the neutral `(x == 5)` case.

`Verify.ignoreIf` can be used as a more precise tool for limiting the search queue, or in a non-heuristic fashion to prune parts of the state space in which it can be shown (via static analysis or manual inspection) that errors cannot occur. The latter approach is used to terminate exploration of infeasible paths when using JPF for symbolic execution [38]. Use of `Verify.ignoreIf` requires considerable caution, as it can result in the model checker returning "true" for properties that do not hold. It is analogous to the `assume` directive available in many other model checkers.

6 Related Work

A wide body of work now exists on the topic of model checking software in real programming languages [3,6,11,24,31,34,35,41,52]. Techniques range from predicate abstraction based approaches [3,6,31] to more direct explorations of executing code without a separate model [24,41,52].

Early work in heuristic model checking applied best-first search to model checking for protocol validation to achieve significant gains over depth-first search [40]. Pageot and Jard [42] discussed using heuristics to guide memory-less search (also known as guided simulation or random walk), and Holzmann noted that this could also be applied to a (potentially) partial search as in explicit-state model checking [32]. Edelkamp, Lafuente, and Leue introduced heuristic search into the SPIN explicit-state model checker [14], suggested a use of heuristic model checking to reduce the size of counterexamples [16], and applied the partial-order reduction to heuristic search [15,13]. This work provides a useful contrast to this paper, in that it concentrates on property and goal-directed heuristics that are sometimes admissible.

We first applied heuristic model checking to Java programs [27] and introduced a new class of heuristics for software model checking [28]. Our previous work and this paper concentrate on structural heuristics over the more studied property and goal-state directed heuristics. Edelkamp and Mehler examined the use of JPF's heuristic framework with goal-directed heuristics, and provide useful commentary on and comparisons with our structural approach [17]. Godefroid and Khurshid applied genetic algorithm techniques rather than the more basic heuristic searches, using heuristics measuring outgoing transitions from a state (similar to the most-blocked heuristic—see Table 2), rewarding evaluations of assertions, and measuring messages exchanged in a security protocol [25]. Musuvathi et al. briefly mention some success in using heuristics to guide a direct exploration of C and C++ code (in a manner similar to that in which JPF explores Java code) [41].

Yang and Dill used a best-first search with BDD-based model checking within the Murϕ tool [54]. Bloem, Ravi and Somenzi used heuristics to reduce the bottlenecks of image computation in symbolic model checking [5]. OBDD-based heuristic search has also been used in AI planning problems closely related to model checking. Edelkamp and Reffel [18] originally proposed an OBDD-based version of the A^* algorithm. Jensen, Bryant and Veloso have developed a significantly improved BDD-based version of A^* [37,36].

Heuristics have also been used for generating test cases [45,51], and model checkers have been used for test case generation [1,2,19,20,23,46]. Friedman et. al. used a Coverage First Search (CFS) related to structural heuristics to generate test suites [21]. Ganai and Aziz used coverage-based techniques to guide a state-space search for control-dependent hardware [22].

7 Conclusions and Future Work

Applying model checking to find errors in real programs is complicated by the size of the state space of such systems. In other fields where search through prohibitively large state spaces is required, such as in AI, the use of heuristics has proven to be invaluable. Here we propose the use of heuristics to guide the search of the JPF model checker for errors in Java programs.

Heuristic search techniques are traditionally used to solve problems where the goal is known and a well-defined measure exists of how close any given state is to this goal. The aim of the heuristic search is to guide the search, using the measure, to achieve the goal as quickly (in the fewest steps) as possible. This has also been the traditional use of heuristic search in model checking: the heuristics are defined with regards to the property being checked. Here we also suggest a complementary approach where the focus of the heuristic search is more on the structure of the state space being searched, in our case the Java program from which the state space is generated.

In addition to property-specific and structural heuristics we also advocate the use of heuristics the user of a model checker can define that are specific to the program being analyzed. In JPF we provide the flexibility to add these heuristics either as external heuristic functions or as annotations of the program being model checked.

Our experimental results show that structural heuristics can make error finding tractable in some systems where unguided searches or searches using property-specific heuristics do not work very well. It is clear that structural heuristics can be useful, and the type of program being explored (programs with complex control flow or that are concurrent) is at least suggestive in making a choice of heuristic.

We believe that the flexibility these various styles of heuristics give the user is an important contribution to the success of model checking as a tool for scalable and efficient error-detection in software systems. However this flexibility is also at the root of the biggest open problem we currently face: which heuristic will work the best in any given circumstance? Our experiments do not really give a clear answer to this question. Ignoring user-defined heuristics, we are already faced by a dizzying array of heuristic options and accompanying parameters—e.g. shall we use a branch counting or the interleaving heuristic, shall we combine them, do we need to add queue-limiting, and what should the limit be? One simple approach is to use the simplest of distributed approaches that computer networks provide and run thousands of model checking runs with different heuristic options each on a different machine on the network with the hope that at least one will produce a positive result.

A more interesting approach would be to *learn* which heuristics would work best. Such an approach could use either the structure of the program code, or the results from failed model checking runs to determine which heuristics to use. For example, in Section 4.3 we show how we can use the results from a race-analysis of the code to guide the search by focusing on the threads that were involved in a race-violation. A further possibility would be to attempt to apply algorithmic learning techniques to finding good parameters for heuristic model checking. We plan to investigate these ideas further in future work.

The development of more structural heuristics and the refinement of those we have presented here is also an open problem. For instance, are there analogous structures to be explored in the data structures of a program to the control structures explored by our branch-coverage heuristics? We imagine that these other heuristics might relate to particular kinds of errors as the interleaving heuristic relates to concurrency errors.

References

1. P. Ammann, P. Black, and W. Majurski. Using Model Checking to Generate Tests from Specications. In *Proceedings of the 2nd IEEE International Conference on Formal Engineering Methods*, 1998.
2. P. Ammann and P. Black. Test Generation and Recognition with Formal Methods. In *Proceedings of the 1st International workshop on Automated Program Analysis, Testing, and Verication*, pages 64–67, 2000.

3. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.
4. B. Beizer. *Software Testing Techniques.* 2nd ed., Van Nostrand Reinhold, New York, 1990.
5. R. Bloem, K. Ravi, and F. Somenzi. Symbolic Guided Search for CTL Model Checking. In *Conference on Design Automation (DAC)*, pages 29–34, 2000.
6. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. To appear in *Proceedings of the 25th International Conference on Software Engineering*, 2003.
7. E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-Guided Abstraction Refinement. In *Proceedings of the 12th Conference on Computer Aided Verification*, pages 154–169, 2000.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
9. E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.
10. 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 *Proceedings of the 23rd International Conference on Software Engineering*, pages 37–46, 2001.
11. J. C. Corbett, M. Dwyer, J. Hatcliff, C. Păsăreanu, Robby, S. Laubach, H. Zheng. Bandera: Extracting Finite-state Models from Java Source Code. In *Proceedings of the 22rd International Conference on Software Engineering*, pages 439–448, 2000.
12. M. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zheng. Tool-supported Program Abstraction for Finite-state Verification. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 177–187, 2001.
13. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed Explicit-state Model Checking in the Validation of Communication Protocols. In *International Journal on Software Tools for Technology Transfer (STTT)*, Springer, 2002.
14. S. Edelkamp, A. L. Lafuente, and S. Leue. Directed Explicit Model Checking with HSF-Spin. In *Proceedings of the 8th International SPIN Workshop on Model Checking of Software*, pages 57–79, 2001.
15. S. Edelkamp, A. L. Lafuente, and S. Leue. Partial Order Reduction in Directed Model Checking. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 112–127, 2002.
16. S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-Directed Model Checking. In *Proceedings of the Workshop of Software Model Checking*, Electrical Notes in Theoretical Computer Science, Elsevier, July 2001.
17. 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)*, August 2003.
18. S. Edelkamp and F. Reffel. OBDDs in Heuristic Search. In *Proceedings of the 22nd Annual German Conference on Advances in Artificial Intelligence (KI-98)*, pages 81–92, 1998.
19. A. Engels, L. Feijs, and S. Mauw. Test Generation for Intelligent Networks Using Model Checking. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398, 1997.
20. J. C. Fernandez, C. Jard, T. Jeron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proceedings of the 8th Conference on Computer Aided Verification*, pages 348–359, 1996.
21. G. Friedman, A. Hartman, K. Nagin, and T. Shiran. Projected State Machine Coverage for Software Testing. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 134–143, 2002.
22. A. K. Ganai and A. Aziz. Efficient Coverage Directed State Space Search. In *International Workshop on Logic Synthesis*, 1998.
23. A Garagantini and C. Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 146–162, 1999.
24. P. Godefroid. VeriSoft: A Tool for the Automatic Analysis of Concurrent Reactive Software. In *Proceedings of the 9th Conference on Computer Aided Verification*, pages 172–186, 1997.
25. P. Godefroid and S. Khurshid. Exploring Very Large State Spaces Using Genetic Algorithms. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 266–280, 2002.
26. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proceedings of the 9th Conference on Computer Aided Verification*, pages 72–83, 1997.
27. A. Groce and W. Visser. Heuristic Model Checking for Java Programs. In *Proceedings of the 9th International SPIN Workshop on Model Checking of Software*, pages 242–245, 2002.
28. A. Groce and W. Visser. Model Checking Java Programs using Structural Heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 12–21, 2002.
29. P. E. Hart, N. J. Nilsson, and B. Raphael. A Formal Basis for Heuristic Determination of Minimum Path Cost. In *IEEE Transactions Syst. Science and Cybernetics*, 4(2):100–107, 1968.
30. K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser and J. White. Formal Analysis of the Remote Agent Before and After Flight. In *Proceedings of the 5th NASA Langley Formal Methods Workshop, June 2000*, 2000.
31. T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Lazy Abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.
32. G. J. Holzmann. Algorithms for Automated Protocol Verification. *AT&T Technical Journal*, 69(2):32–44, Feb. 1990, pages 32–44. Special Issue on Protocol Testing, Specification, and Verification.
33. G. J. Holzmann and Doron Peled. The State of SPIN. In *Proceedings of the 8th Conference on Computer Aided Verification*, pages 385–389, 1996.
34. G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. In *Bell Labs Technical Journal*, 5(2);72–87 April-June 2000

35. R. Iosif and R. Sisto. dSPIN: A Dynamic Extension of SPIN. In *Proceedings of the 6th International SPIN Workshop on Model Checking of Software*, pages 261–276, 1999.
36. R. M. Jensen, R. E. Bryant, and M. M. Veloso. An Efficient BDD-based A^* Algorithm In *AIPS-02 Workshop on Planning via Model Checking*, 2002.
37. R. M. Jensen, R. E. Bryant, and M. M. Veloso. SetA^*: An Efficient BDD-Based Heuristic Search Algorithm. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-02)*, pages 668-673, 2002.
38. S. Khurshid, C. S. Păsăreanu, and W. Visser. Generalized Symbolic Execution for Model Checking and Testing. To appear in *Tools and Algorithms for Construction and Analysis of Systems*, 2003.
39. B. Korel. Automated Software Test Data Generation. In *IEEE Transaction on Software Engineering*, 16(8):870-879, August 1990.
40. F. J. Lin, P. M. Chu, and M. T. Liu. Protocol Verification Using Reachability Analysis: The State Space Explosion Problem and Relief Strategies. *ACM*, 126–135, 1988.
41. M. Musuvathi, D. Park, A. Chou, D. Engler, and D. Dill. CMC: A Pragmatic Approach to Model Checking Real Code. In *Proceedings of the Fifth Symposium on Operating System Design and Implementation*, 2002.
42. J. M. Pageot and C. Jard. Experience in Guiding Simulation. In *Proceedings of the VIII-th Workshop of Protocol Specification, Testing, and Verification*, 1988.
43. C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *International Journal on Software Tools for Technology Transfer (STTT)*, 5(1); 34–48, November 2003.
44. J. Penix, W. Visser, E. Engstrom, A. Larson and N. Weininger. Verification of Time Partitioning in the DEOS Scheduler Kernel. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 488–497, 2000.
45. A. Pretschner. Classical search strategies for test case generation with Constraint Logic Programming. In *Proceedings of the Workshop on Formal Approaches to Testing of Software*, pages 47–60, 2001.
46. S. Rayadurgam and M. P. Heimdahl. Coverage Based Test-Case Generation using Model Checkers. In *Proceedings of the 8th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systsems*, pages 83–93, 2001.
47. RTCA Special Committee 167. Software Considerations in Airborne Systems and Equipment Certification. Technical Report DO-178B, RTCA, Inc., Dec. 1992.
48. H. Saidi. Modular and Incremental Analysis of Concurrent Software Systems. In *Proceedings of the 14th IEEE International Conference on Automated Software Engineering (ASE)*, pages 92–101, 1999.
49. S. Savage, M. Burrows, G. Nelson, and P. Sobalvarro. Eraser: A Dynamic Data Race Detector for Multithreaded Programs. In *ACM Transactions on Computer Systems*, 15(4):391–411, November 1997.
50. R. Sun and C. Sessions. Learning plans without a priori knowledge. In *Adaptive Behavior*, 8(3/4), 2001.
51. N. Tracey, J. Clark, K. Mander, and J. McDermid. An Automated Framework for Structural Test-Data Generation. In *Proceedings of the 13th IEEE International Conference on Automated Software Engineering (ASE)*, pages 285–288, 1998.
52. W. Visser, K. Havelund, G. Brat and S. Park. Model Checking Programs. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering (ASE)*, pages 3–11, 2000.
53. W. Visser, S. Park, and J. Penix. Using Predicate Abstraction to Reduce Object-Oriented Programs for Model Checking. In *Proceedings of the 3rd ACM SIGSOFT Workshop on Formal Methods in Software Practice*, 2000.
54. C. Han Yang, D. L. Dill. Validation with Guided Search of the State Space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.