Model Checking Java Programs using Structural Heuristics

Alex Groce
School of Computer Science, Carnegie Mellon University
agroce@cs.cmu.edu

Willem Visser
RIACS/NASA Ames Research Center
wvisser@riacs.edu

Keywords
model checking, testing, heuristics, coverage metrics

ABSTRACT

We describe work introducing heuristic search into the Java PathFinder model checker, which targets Java bytecode[1]. Rather than focusing on heuristics aimed at a particular kind of error (such as deadlocks) we describe heuristics based on a modification of traditional branch coverage metrics and other *structural* measures, such as thread inter-dependency. We present experimental results showing the utility of these heuristics, and argue for the usefulness of *structural heuristics* as a class.

1. INTRODUCTION

There has been recent interest in model checking software written in real programming languages [3, 10, 15, 24, 25, 33] and in using heuristics to direct exploration in explicit-state model checkers [12, 35]. Because heuristic-guided search is clearly directed at finding errors rather than verifying the complete correctness of software, the connections between model checking and testing are made particularly clear when these ideas are combined. In this paper we present one fruitful product of the intersection of these fields and show how to apply it to finding errors in programs.

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 [8], first in hardware verification, and now, increasingly, in software verification [3, 10, 21]. 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.

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. Heuristic model checking usually 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. However, by using measurements of the exploration of a program's structure (in particular, its branching structure or thread interdependency structure), we believe a model checker can often improve its ability to find errors in programs. 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.

In this paper we explore heuristic model checking of software written in the Java programming language and use heuristics based on coverage measurements derived from the world of software testing. We introduce the notion of *structural heuristics* to the classification of heuristics used in model checking, and present (and describe our motivations in developing) successful and novel heuristics from this class.

The paper is organized as follows. Section 2 describes heuristic model checking, examines related work, and introduces the various search algorithms we will be using. Section 3 briefly presents the Java PathFinder model checker and the implementation of heuristic search. The new heuristics are defined and described in detail in section 4, which also includes experimental results. We present 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

[1] We present the basic heuristic framework and discuss the creation of user defined heuristics in a tool paper elsewhere [18].

```
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 then terminate
            $f$ = $h(S')$
            store ($S'$, $f$) in $Q$
```

Figure 1: Algorithm for *best-first* search.

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

Figure 2: Algorithm for *beam* search.

resulting counterexamples will often be shorter than ones produced by the depth-first search based algorithms traditionally used in explicit-state model checkers.

The growing body of literature on model checking using heuristics largely concentrates on heuristics tailored to find a certain kind of error [12, 16, 22, 26, 35]. Common heuristics include measuring the lengths of queues, giving preference to blocking operations [12, 26], and using a Hamming distance to a goal state [14, 35]. Godefroid and Khurshid apply genetic algorithm techniques rather than the more basic heuristic searches, using heuristics measuring outgoing transitions from a state (similar to our most-blocked heuristic – see Table 1), rewarding evaluations of assertions, and measuring messages exchanged in a security protocol [16]. Heuristics can also 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. However, we suggest 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. The heuristic used in FLAVERS would be an example of the latter [9]. We concentrate primarily on structural heuristics, and will further refine this notion after we have examined some of our heuristics.

Heuristics have also been used for generating test cases [29, 32], and, furthermore, a model checker can be used for test case generation [1, 2]. Our approach is not only applicable to test case generation, but applies coverage metrics used in testing to the more usual model checking goal of finding errors in a program.

2.1 Search Algorithms

A number of different search algorithms can be combined with heuristics. The simplest of these 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 [19] is similar, except that like Dijkstra's shortest paths algorithm, it adds the length of the path to S' to f. 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 our purposes, the shortest counterexample). 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).

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, of course, 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 examined are admissible, A^* lacks a theoretical optimality, and is generally less efficient than best-first search. Our 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.

More specifically, the introduction of queue-limiting to heuristic search for model checking appears to be genuinely novel, and 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 over-conservative in the context of heuristic search [13]. 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 4.1.1 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 [33]. 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 [25] that extends SPIN [23] 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 class (called `Verify`) allow nondeterminism to be expressed (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 [14].

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

4. STRUCTURAL HEURISTICS

We consider the following heuristics to be *structural heuristics*: that is, they are intended to find errors, but are not targeted specifically at particular assertion statements, invariants, or deadlocks. Rather, they explore some structural aspect of the program (branching structure or thread-interdependence).

4.1 Code Coverage Heuristics

The code coverage achieved during testing is a measure of the adequacy of the testing, in other words 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 [30].

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 worth, 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

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).

Figure 3: Our basic branch-coverage heuristic.

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 4.1.1). Instead, a slightly more complex heuristic proves successful (Figure 3).

The motivation behind this 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. Our heuristic behaves in an essentially breadth-first manner unless a path is actually increasing coverage. By default, our system 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, because the frontier is much deeper along paths which have previously increased coverage, we still advance exploration of structurally interesting paths.

Our heuristic delays exploration of repetitive portions of the state space (those that take the same branches repeatedly). If a nondeterminisic choice determines how many times to execute a loop, for instance, our heuristic 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. We thus achieve deeper coverage of the structure and examine 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, we will return to explore further iterations of the first loop before executing the latter loop more than once.

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 (see Figure 3). 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.

4.1.1 Experimental Results

We will refer to a number of heuristics in our experimental results (Table 1). 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. Some results are for an A^* or *beam* search, and this is also noted.

Search/Heuristic	Definition
branch	The basic branch-coverage heuristic.
%-coverage	Measures the percentage of branches covered. States with higher coverage receive better values.
BFS	A breadth-first search
DFS	A depth-first search. (depth n) indicates that stack depth is limited to n.
most-blocked	Measures the number of blocked threads. More blocked threads result in better values.
interleaving	Measures the amount of interleaving of threads. See section 4.2.
random	Uses a randomly assigned heuristic value.

Table 1: Heuristics and search strategies.

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) [28]. 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 4.3). Our results (Table 2) 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 our 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.

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. We calculate this heuristic by keeping a (possibly limited in size) history of the threads scheduled on each path (Figure 4).

- 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: Our basic interleaving heuristic.

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 [20]. Our results (Table 3) 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.

Our experiments (here and in other examples not presented in the interest of space) indicate that while A^* and beam-search can certainly perform well at times, they generally do not perform as well as best-first search. Our heuristics 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.

Finally, for the dining philosophers (Table 4), we show that our 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

Search/Heuristic	Time(s)	Memory(MB)	States Explored	Length	Max Depth
branch (path)	60	92	2,701	136	139
branch (path)(A^*)	59	90	2,712	136	139
branch (global)	60	91	2,701	136	139
branch (global)(A^*)	59	92	2,712	136	139
%-coverage (path)	FAILS	-	20,215	-	334
%-coverage (path)(A^*)	FAILS	-	18,141	-	134
%-coverage (global)	FAILS	-	20,213	-	334
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					
branch(path)	FAILS	-	15,964	-	125
branch(path) (A^*)	FAILS	-	15,962	-	125
branch(global)	FAILS	-	15,964	-	125
branch(global) (A^*)	FAILS	-	15,962	-	125

Table 2: 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 **Memory** 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).

Search/Heuristic	Time(s)	Memory(MB)	States Explored	Length	Max Depth
branch (path) (queue 40)	FAILS	-	1,765,009	-	12,092
branch (path) (queue 160)	FAILS	-	1,506,725	-	5,885
branch (path) (queue 1000)	132	290	845,263	136	136
branch (global) (queue 40)	FAILS	-	1,758,416	-	12,077
branch (global) (queue 160)	FAILS	-	1,483,827	-	1,409
branch (global) (queue 1000)	FAILS	-	1,509,810	-	327
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
interleaving	FAILS	-	378,068	-	81
interleaving (queue 5)	15	17	38,449	913	913
interleaving (queue 40)	116	184	431,752	869	869
interleaving (queue 160)	908	501	1,287,984	869	870
interleaving (queue 1000)	FAILS	-	745,788	-	177
interleaving (A^*)	FAILS	-	369,166	-	81
interleaving (queue 5) (A^*)	13	19	43,172	912	912
interleaving (queue 40) (A^*)	77	129	306,285	865	867
interleaving (queue 160) (A^*)	FAILS	-	1,309,561	-	789
interleaving (queue 1000) (A^*)	FAILS	-	1,836,675	-	273
interleaving (queue 5) (beam)	14	16	35,514	927	927
interleaving (queue 40) (beam)	91	113	238,945	924	924
interleaving (queue 160) (beam)	386	418	1,025,595	898	898
interleaving (queue 1000) (beam)	FAILS	-	1,604,940	-	365
most-blocked	7	33	7,537	158	169
most-blocked (queue 5)	FAILS	-	922,433	-	27,628
most-blocked (queue 40)	FAILS	-	913,946	-	4,923
most-blocked (queue 160)	FAILS	-	918,575	-	1,177
most-blocked (queue 1000)	6	10	7,537	158	169
most-blocked (A^*)	FAILS	-	631,274	-	61
most-blocked (queue 5) (A^*)	FAILS	-	935,796	-	16,189
most-blocked (queue 40) (A^*)	FAILS	-	960,259	-	1,907
most-blocked (queue 160) (A^*)	FAILS	-	989,513	-	555
most-blocked (queue 1000) (A^*)	FAILS	-	1,138,920	-	165

Table 3: Experimental results for the Remote Agent system.

Search/Heuristic	Size	Time(s)	Memory(MB)	States Explored	Length	Max Depth
branch (path)	8	FAILS	-	374,152	-	41
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
most-blocked	8	FAILS	-	310,317	-	285
most-blocked (queue 5)	8	17,259	378	891,177	78,353	78,353
most-blocked (queue 40)	8	10	7	13,767	273	273
most-blocked (queue 160)	8	10	12	25,023	172	172
most-blocked (queue 1000)	8	46	59	123,640	254	278
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
most-blocked (queue 5)	16	FAILS	-	802,526	-	36,443
most-blocked (queue 40)	16	38	69	101,576	1,008	1,008
most-blocked (queue 160)	16	FAILS	-	799,453	-	2,071
most-blocked (queue 1000)	16	FAILS	-	791,073	-	702
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
most-blocked (queue 40)	32	FAILS	-	463,414	-	2,251
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

Table 4: Experimental results for dining philosophers.

threads. We have only reported, 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 1), since if all threads are blocked we have a deadlock in a Java program. Whereas a structural 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 1). 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.

Godefroid and Khurshid apply their genetic algorithm techniques to a very similar implementation of the dining philosophers (written in C rather than Java) [16]. 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). Our results suggest that the differences may be as much a result of the heuristics used (something like most-blocked vs. our interleaving heuristic) as the genetic search itself. Application of our heuristics in different search frameworks is an interesting avenue for future study.

4.3 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 [7, 11, 17, 34]. JPF supports two forms of over-approximation: predicate abstraction [34] and type-based abstractions (via the BANDERA tool) [11]. 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, 6, 21, 27, 31].

JPF uses a novel technique for the elimination of spurious errors called *choose-free* search [27]. 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 [11, 27]. 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 [34] and type-based abstraction [11]. 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 [11]. 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.

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 [9].

5. CONCLUSIONS AND FUTURE WORK

Heuristic search techniques are traditionally used to solve problems where the goal is known and a well-defined measure exists of how close one is to this goal. The aim of the heuristic search is to guide the search, using the measure, to achieve the goal as quickly (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 advocate 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.

We do not believe that structural heuristics should replace property-specific heuristics, but rather propose that they be used as a complementary approach. Furthermore, since the testing domain has long used the notion of structural code coverage, it seems appropriate to investigate similar ideas in the context of structural heuristics during model checking. Here we have shown that for a realistic example (DEOS) a heuristic based on branch coverage (a relatively weak structural coverage measure) gives encouraging results. It is worth noting that a much stronger coverage measure (MC/DC) did not "help" in uncovering the same error during testing (i.e. 100% coverage was achieved but the bug was not found). We conjecture that the use of code coverage during heuristic model checking can lead to classes of errors being found that the same coverage measures during testing will not uncover. For example, branch coverage is typically of little use in uncovering concurrency errors, but using it as a heuristic in model checking will allow the model checker to evaluate more interleavings which might lead to an error (branch coverage found the deadlock in the Remote Agent example, whereas traditional testing failed[2]).

There are a number of possible avenues for future work. As our experimental results make clear, a rather daunting array of parameters is available when using heuristic search – at the very least, a heuristic, search algorithm, and queue size must be selected. We hope to explore the practicalities of selecting these options, gathering more experimental data to determine if, for instance, as it appears, proper queue size limits are essential in checking programs with a large number of threads. A further possibility would be to attempt to apply algorithmic learning techniques to finding good parameters for heuristic model checking.

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.

6. 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] 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.

[2] Although we don't know whether branch coverage was used as a measure during the original testing, the structure of the code indicates that this coverage would have been achieved with any reasonable test set

[7] 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.

[8] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 2000.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] T. A. Henzinger, R. Jhala, R. Majumdar and G. Sutre. Lazy Abstraction. In *ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, 2002.

[22] 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.

[23] G. J. Holzmann and Doron Peled. The State of SPIN. In *Proceedings of the 8th Conference on Computer Aided Verification*, pages 385–389, 1996.

[24] G. J. Holzmann and M. H. Smith. Automating Software Feature Verification. In *Bell Labs Technical Journal*, 5(2);72–87 April-June 2000

[25] 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.

[26] 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.

[27] C. S. Păsăreanu, M. B. Dwyer, and W. Visser. Finding Feasible Counter-examples when Model Checking Abstracted Java Programs. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 284–298, 2001.

[28] 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.

[29] 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.

[30] RTCA Special Committee 167. Software considerations in airborne systems and equipment certification. Technical Report DO-178B, RTCA, Inc., Dec. 1992.

[31] 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.

[32] 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.

[33] 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.

[34] 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.

[35] C. Han Yang, D. L. Dill. Validation with Guided Search of the State Space. In *Conference on Design Automation (DAC)*, pages 599–604, 1998.