Heuristic-Guided Counterexample Search in FLAVERS*

Jianbin Tan, George S. Avrunin, Lori A. Clarke, Shlomo Zilberstein
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003, USA
{tjb, avrunin, clarke, shlomo}@cs.umass.edu

Stefan Leue
Department of Computer and Information Science
University of Konstanz
D-78457 Konstanz, Germany
Stefan.Leue@inf.uni-konstanz.de

ABSTRACT

One of the benefits of finite-state verification (FSV) tools, such as model checkers, is that a counterexample is provided when the property cannot be verified. Not all counterexamples, however, are equally useful to the analysts trying to understand and localize the fault. Often counterexamples are so long that they are hard to understand. Thus, it is important for FSV tools to find *short* counterexamples and to do so *quickly*. Commonly used search strategies, such as breadth-first and depth-first search, do not usually perform well in both of these dimensions. In this paper, we investigate heuristic-guided search strategies for the FSV tool FLAVERS and propose a novel two-stage counterexample search strategy. We describe an experiment showing that this two-stage strategy, when combined with appropriate heuristics, is extremely effective at quickly finding short counterexamples for a large set of verification problems.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification—*formal methods, model checking, validation*

General Terms

Verification

*This research was partially supported by the National Science Foundation under grant CCR-0205575, by the U.S. Army Research Laboratory and the U.S. Army Research Office under agreement DAAD190110564, and by the U.S. Department of Defense/Army Research Office under agreement DAAD190310133.
Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation, the U. S. Army Research Office or the U. S. Department of Defense/Army Research Office.

Keywords

Heuristic search, counterexamples, FLAVERS

1. INTRODUCTION

Finite-state verification approaches attempt to prove properties about a model of a system. These approaches are not as general as theorem-proving based verification, but they are usually easier to use. In addition, when it is found that a property does not hold, most FSV tools provide a counterexample, a trace through the model that illustrates how the property can be violated. The counterexample is important to analysts since it contains clues about the cause of the violation. By studying the counterexample, analysts can usually determine if the trace represents an actual violation or a false positive. For an actual violation, analysts will correct the system, its corresponding model, or the property and then rerun the verification. For a false positive result, analysts use the information from the counterexample to try to determine how to improve the precision of the model so that such spurious violations will be eliminated on subsequent verification attempts. In this iterative process of correcting or improving the system, model, or property, short counterexamples are usually preferred since they are easier for analysts to understand. It is, of course, also desirable that the verification be relatively fast so that analysts do not have to wait for long periods of time before learning the verification results. Therefore, it is desirable that FSV tools return results *quickly* and, when a property can not be verified, provide a *short* counterexample.

Most FSV tools, such as SPIN [18] and FLAVERS [6, 8], represent the verification problem as a search problem over the model of the system. This model is usually a graph representation of the system, optimized for the specific property to be verified. Accordingly, the counterexample is a path through the model. Commonly used search strategies, such as breadth-first search (BFS) and depth-first search (DFS), may be used in this context. Previous work [5] showed that for FLAVERS the BFS strategy, which always finds the shortest counterexample as long as it does not run out of memory, tends to have disappointing execution performance, whereas the DFS strategy tends to find a counterexample quickly but this counterexample tends to be very long. Thus, for FLAVERS neither of these two strategies satisfies both the time and length requirements. We would expect similar results for other FSV tools.

In this paper we report on our efforts to find a heuristic-

guided search strategy that will tend to return short counterexamples quickly. In a heuristic graph search algorithm, problem-specific knowledge, typically based on a goal node, is used to associate an estimated metric with each node. This metric is then used to discriminate among the nodes during the search. The heuristics we considered are based on the structure of the model of the system and on the property to be verified. Moreover, we propose a novel two-stage search strategy that is designed for properties represented by automata having a special form. All safety properties can be represented by such automata. We demonstrate that the two-stage search strategy, combined appropriately with the heuristics, is particularly effective. The evaluation is conducted using the FLAVERS/Ada toolset applied to a set of Ada tasking programs. Although our experimental evaluation is based on analysis of Ada programs with FLAVERS, we believe that the results provide insights that could be applied to other FSV techniques and other programming languages.

In the next section of this paper, we provide an overview of FLAVERS so that the reader can understand the intuition behind the heuristics that we investigated. The third section describes these heuristics and the two-stage search strategy. Section 4 discusses our experimental methodology, and Section 5 presents the experimental results. Section 6 describes related work, and we conclude in Section 7 with a summary of the results and a discussion of future work.

2. FLAVERS OVERVIEW

FLAVERS/Ada is an event-based FSV tool that can check if all possible executions of an abstract model of a system are consistent with a user-specified property. The property represents desirable (or undesirable) sequences of events that should occur on all (or none) of the executions of the system. An event is typically some syntactically recognizable executable action in the system, such as a method call or task synchronization. The property must be represented in a notation that can be translated into a finite-state automaton (FSA) representation, where a transition represents the occurrence of an event. For example, Figure 1(a) is a property FSA, involving events **T2.lock** and **T2.unlock**, for the system of two communicating tasks described in Figure 1(b). This property specifies that two instances of event **T2.lock** cannot occur consecutively

The model of the system used in FLAVERS, called a *Trace Flow Graph* (TFG), is automatically derived from the system description (e.g., the source code). Since the property is described in terms of sequences of events, the TFG must appropriately represent the control flow among these events in the system. For a sequential system, this model would be an annotated control flow graph, where the nodes in the graph correspond to the execution of the action associated with an event. For simplicity, we create the model so that at most one event is associated with a node. If a node does not have any events associated with it and does not affect the flow of control for any nodes that do, it may be removed from the model. If the events of a property occur infrequently in the system, the resulting model is usually very small. Thus, it is generally practical to inline all method calls.[1]

To model a concurrent Ada system, each task is represented by an annotated control flow graph, as described above, and then some modifications are made to represent the synchronization and the interleaving of events. Specifically, communication nodes are created that conceptually "merge" the nodes that represent the rendezvous between two tasks and May Immediately Precede edges (MIP edges) (created using the algorithm of [19]) are used to represent the potential interleavings of events in different tasks.

Formally, a TFG is a labeled directed graph, $G = (N, E, n_{initial}, n_{final}, \mathcal{A}_G, L)$, where N is a finite set of nodes, $E \subseteq N \times N$ is a set of directed edges, $n_{initial}, n_{final} \in N$ are initial and final nodes of the TFG, $\mathcal{A}_G$ is an alphabet of event labels associated with the TFG, and $L : N \to \mathcal{A}_G \cup \{\emptyset\}$ is a function mapping nodes to their labels or to the null event.

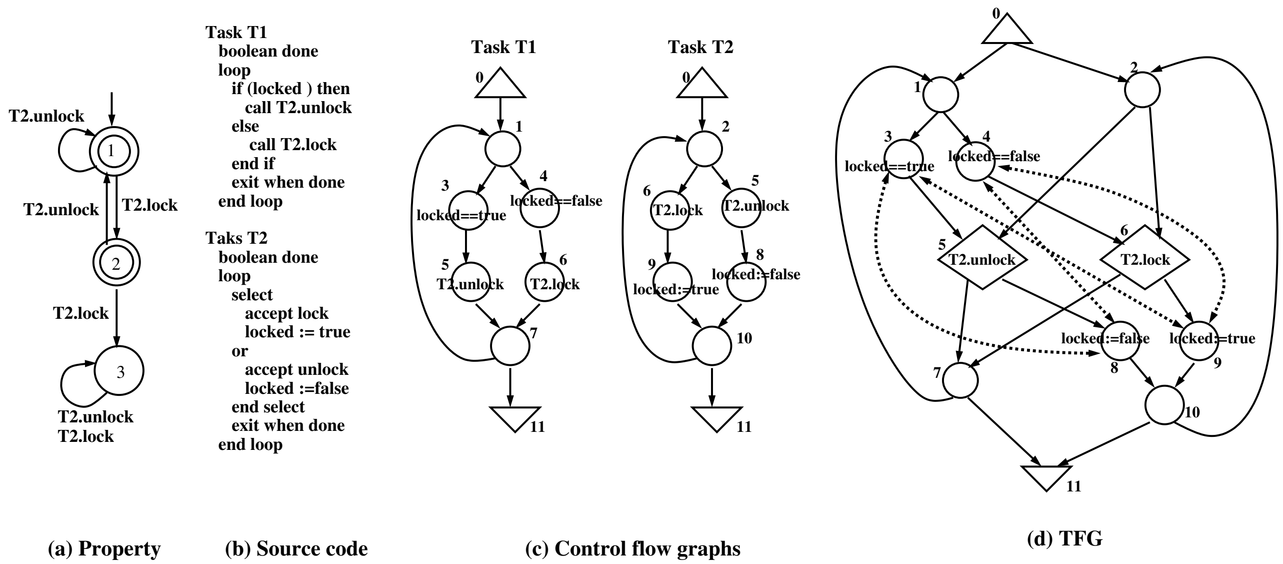
Figure 1(c) shows the control flow graphs for the system in Figure 1(b), and Figure 1(d) gives the TFG for this system. In Figure 1(d), the diamond-shaped nodes represent communication nodes, which model the Ada rendezvous, and the triangular nodes represent the initial and final nodes. The dashed edges in Figure 1(d) represent MIP edges.

The TFG model is *conservative*, meaning that each sequence of events that could occur during the execution of the system corresponds to a path in the TFG that results in traversing the same sequence of events. Therefore, when all event sequences in the model are accepted by the property, it is safe to conclude that the property holds on the original system. On the other hand, if there is a sequence of events in the model that is not accepted by the property, it may be an indication of an error in the system (or in the property), or it may be that the path in the model that traverses the sequence of events does not correspond to any real execution in the system and thus is infeasible. Thus, the model is imprecise in that it may overapproximate the behavior of the system.

One of the strengths of FLAVERS is that analysts can incrementally improve precision by augmenting the model with constraints that may eliminate at least some of the infeasible paths. These constraints are represented as FSAs, and they specify the sequences of events that must occur during any execution of the system. When a sequence of events in the model violates any constraint, the corresponding path is considered to be infeasible and thus is eliminated from consideration.

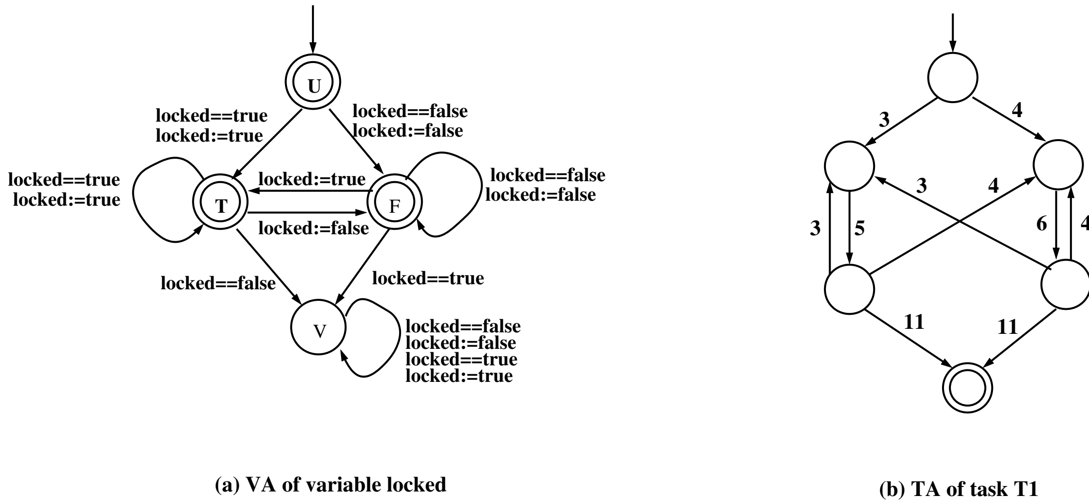
Three kinds of constraints are commonly used in FLAVERS: *Context Automata* (CAs) are used to model the environment; *Variable Automata* (VAs) are used to model the value of variables; and *Task Automata* (TAs) are used to model execution traces for tasks. CAs are usually specified by analysts according to knowledge about the environment in which the system will be executed. VAs and TAs, however, can usually be generated automatically by FLAVERS depending on the type of the variable or on the control flow graph of a task, respectively. Figure 2(a) shows the VA for the boolean variable **locked** in the system of Figure 1. This VA specifies that when the variable **locked** is *true*(or *false*), the event "locked==false" (or "locked==true") that corresponds to a test of **locked** returning the opposite value can not occur. Figure 2(b) shows the TA for task **T1** in the system of Figure 1. This TA is based on the control flow graph of the task **T1** shown in Figure 1, with two changes. First, those nodes in the control flow graph that do not have an associated event are not modeled in the TA. Second, the transition label in the TA is the index of the correspond-

[1]FLAVERS currently does not handle recursive calls.



(a) Property **(b) Source code** **(c) Control flow graphs** **(d) TFG**

Figure 1: A simple example



(a) VA of variable locked **(b) TA of task T1**

Figure 2: VA and TA examples

ing state in the control flow graph. Each state in the TA may be thought of as a program counter value for that task. Note that each TA has only one accepting state, which corresponds to the termination of the task.

Given a *subject* that includes a TFG, a property, and a set of constraints, FLAVERS uses one of two different algorithms to check if there is an event sequence in the TFG that is accepted by every constraint but violates the property. The *State Propagation* algorithm [8] is a data-flow analysis fix-point algorithm. The *Find Path* algorithm is a typical search algorithm. As shown in [5], the State Propagation algorithm tends to be more efficient than the Find Path algorithm when the property turns out to hold. In those cases where the property is found to be violated, however, the Find Path algorithm is more efficient. In addition, it is easier to generate a counterexample for the Find Path algorithm than the State Propagation algorithm. Therefore, the Find Path algorithm is probably a better platform for exploring heuristics for counterexample search and thus is the only approach considered further in this paper.

As shown in Figure 3, the Find Path algorithm performs a search on a *node-tuple graph*, whose vertices are pairs consisting of a node from the TFG node and a tuple or vector of states, one from the property FSA and one from each constraint FSA. In the node-tuple graph, node-tuple $\langle n', t' \rangle$ with TFG node n' and tuple t' is a successor of $\langle n, t \rangle$ if and only if n' is a successor of n in the TFG and t' is computed by applying the event labeling n' to each state in t.

The algorithm starts with an initial node-tuple that is composed of the initial TFG node and the start state of each FSA and builds the node-tuple graph on-the-fly while it conducts a search for a violation of the property. A violation of the property is found if a node-tuple is encountered where the node is the final node of the TFG and, for the states in the tuple, the property state is non-accepting and all constraint states are accepting. As soon as such a *vio-*

lating node-tuple is discovered, the algorithm returns an *inconclusive* result, meaning that the property is violated. In addition, a counterexample that is a path in the node-tuple graph from the initial node-tuple to the violating node-tuple is provided by tracing back through the ancestors of the violating node-tuple. If all the node-tuples reachable from the initial node-tuple are examined and no violating node-tuple is found, the algorithm returns a *conclusive* result, meaning that the property holds. Since the TFG and all FSAs are finite, and each node-tuple is visited only once, the algorithm will always terminate.

```
Add the initial node-tuple in the worklist W
While W is not empty
  Remove a node-tuple n from W
  For each successor s of n
    If s has been encountered before
      continue
    Else
      Set n to be the parent of s
      If s is a violating node-tuple
        Generate the counterexample by
        tracing back the parents of s
        Return INCONCLUSIVE
      Else
        Add s to W
      End If
    End If
  End For
End While
Return CONCLUSIVE
```

Figure 3: The Find Path algorithm

3. HEURISTICS CONSIDERED

Generally speaking, a heuristic-guided search strategy uses an evaluation function, $f(n)$, to determine the order in which the nodes in the search graph are selected for expansion. The evaluation function measures the distance to a goal node based on a heuristic function, $h(n)$, which gives the estimated shortest distance from the given node n to a goal node. To make Find Path a heuristic search algorithm, we can simply use a priority queue, a data structure that sorts the node-tuples according to their f-values, as the worklist.

We considered two heuristic search algorithms, the Best First (BF) algorithm and the Weighted A* (WA*) algorithm [15]. In the BF algorithm, the evaluation function is just the heuristic function, which considers the estimated distance between the current node and a goal node. Consequently, the BF algorithm is a goal-oriented DFS-like algorithm and thus is likely to find a goal node quickly. The WA* algorithm is a generalized version of the widely known A* algorithm [16]. In this algorithm, the evaluation function is defined to be $f(n) = g(n) + w * h(n)$, where $h(n)$ is a heuristic function, $g(n)$ is the distance from the initial node to the node n, and w is a parameterized weight. Unlike the BF algorithm, which estimates the distance to the goal from the *current* node, WA* measures the estimated distance from the *initial* node to a goal node. WA* is more BFS-like, in that it will backtrack to an earlier node, and thus tends to find a short path.

The value of the weight w in the WA* algorithm is usually determined by experimentation. The weight actually represents a trade-off between the search time and the length of the returned path. When the weight is 1, the WA* algorithm becomes identical to the A* algorithm. The A* algorithm is guaranteed to find the shortest path to a goal node provided that the heuristic function is admissible, which means it never overestimates the distance to that goal. On the other hand, when the weight is large, the WA* algorithm returns the same path returned by the BF algorithm, which tends to find a goal node quickly, as described above. Selecting a weight, therefore, provides control over the trade-off between the search time and the length of the path to the goal node.

When applying the WA* algorithm to find counterexamples in FLAVERS, we define $g(n)$ to be the length of the *first* path that reaches the node-tuple n from the initial node-tuple, as described in Figure 3. This definition of $g(n)$ is different from the standard one, which updates $g(n)$ whenever a shorter path to reach node n is discovered. We experimentally evaluated both definitions and found that the one based on the first path makes the returned counterexample slightly longer but almost always reduces the amount of time; we therefore use this definition of $g(n)$ throughout this paper. In addition, as shown in Figure 3, we generate a counterexample as soon as a violating node-tuple is encountered, whereas in the A* algorithm the solution is generated when only a goal node is selected for expansion. This conservative approach of the A* is not used since we are not seeking the optimal solution, only a good solution.

The heuristic function is obviously a key element in a heuristic search algorithm. To estimate the distance between a given node and a goal node, the heuristic function is usually based on some aspects of the goal node. In the context of the counterexample search in FLAVERS, a goal node is a violating node-tuple. In this section, we describe two heuristics, the TA heuristic and the trap heuristic, each of which focuses on different aspects of the violating node-tuple. The TA heuristic uses the TA states in the node-tuple to calculate the estimate, while the trap heuristic uses the property state in the node-tuple. After describing these two heuristics, we present a two-stage search strategy that is also based on the property to be verified.

3.1 The TA Heuristic

Recall that in a violating node-tuple, all constraint states are accepting. Since each TA constraint has only one accepting state, this is the state associated with the violating node-tuple for that TA. Based on this observation, the TA heuristic [5] estimates the shortest distance from a given node-tuple to a violating node-tuple by summing the shortest distance from the current state to the accepting state for each TA. The intuition behind the TA heuristic is that it provides an estimate of the smallest number of node-tuples that need to be visited before reaching a violating node-tuple. Since this heuristic never overestimates the distance to a violating node-tuple, it is admissible.[2]

There are other aspects of a node-tuple besides the TA states that we do not consider in formulating this heuristic function, such as the current TFG node, current property state, and other current constraint states. Experimentally, we found that these aspects are not good predictors of the path length. The TFG is not a good predictor because it in-

[2]To make sure the TA heuristic is admissible, we must assure that each communication node, representing a rendezvous between two tasks, is only counted once in the estimate.

cludes infeasible paths that violate the TA constraints. Unlike the TA constraints that have only one goal state (the only accepting state), each property or non-TA constraint may have several goal states (any non-accepting state for the property and any accepting state for the constraint). Thus, the distance from the current property state or non-TA constraint state to its goal state is also not a good predictor of the path length since multi-goals tend to make these distances indifferently small.

We also considered using these other aspects to break ties when the TA heuristic returned identical minimal values for more than one node-tuple. In our experiments, such tie-breaking provided only slightly better results than arbitrary choice and thus will not be discussed further in this paper.

As shown in Section 5, the simple TA heuristic is a relatively good predictor of path length and can be used in both the WA* algorithm, yielding an algorithm we call WA_{TA}, and the BF algorithm, yielding an algorithm we call BF_{TA}.

3.2 The Trap Heuristic

A trap state in an FSA is a non-accepting state that does not have any outgoing transitions except self-loops. It is easy to see that once the property enters a trap state, it is violated and, unlike other non-accepting states, there is no way to recover from the violation. Consequently, if there is more than one trap state in a property, these states can be combined into a single trap state. A node-tuple whose property state is the trap state is called a *trap node-tuple*. When a trap node-tuple is encountered, it is a good indication that the property might be violated. Violation is still uncertain, however, until a path is found from this trap node-tuple to a violating node-tuple that does not violate any constraint. We note that any safety property can be represented by a prefix-closed finite automaton [1], and therefore that any safety property can be represented by an automaton with a single trap state.

Because of the likelihood of finding a violation, it seems reasonable to use the trap state as the goal of the heuristic function. We could define a simple heuristic, similar to the TA heuristic, that computes the shortest distance between the current property state and the trap state in the property FSA.[3] The evaluation function would use this heuristic to select the node-tuple on the worklist that causes a property transition to the state with the lowest heuristic value. The problem with this simple heuristic is that, in most property FSAs, most states have at least one transition to the trap state. As a result, these states have the same metric value, namely 1. Our trap heuristic solves this problem by also considering the number of transitions to the trap state. In the trap heuristic, if a state does not have any transitions to the trap state, then its metric value is simply the shortest distance to the trap state. For a state that has transitions to the trap state, the metric value is equal to $1 + 1/n$, where n is the number of the transitions to the trap state. This makes the metric smaller on states with more transitions to the trap state. The metric value for the trap state is always 0. Figure 4 gives an example of the calculation of the trap heuristic.

[3]As mentioned above, the shortest distance to a non-accepting state of the property FSA is not a good predictor of path length because there is usually more than one non-accepting state. This simple heuristic, however, restricts the goal to a single state and thus does not have this problem.

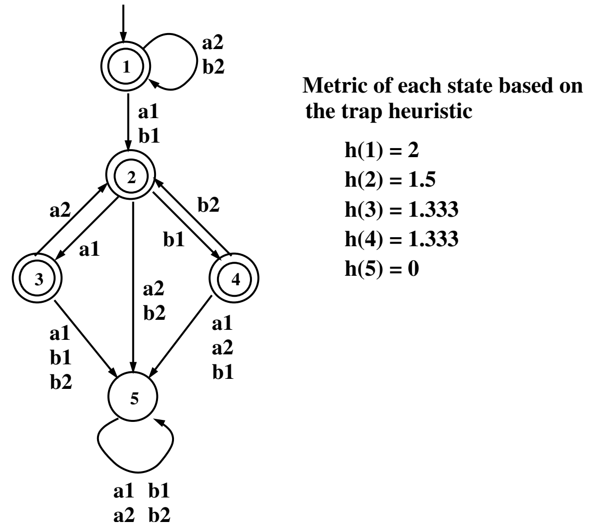


Figure 4: A trap heuristic example

Since the trap heuristic considers the distance within the property FSA, it does not make sense to use it in the WA* algorithm, where $g(n)$ gives the distance in the node-tuple graph. Therefore, the trap heuristic is only used in the BF algorithm, yielding an algorithm we call BF_{trap}.

3.3 A Two-stage Search Strategy

For properties with a trap state, it is reasonable to consider a two-stage search strategy. In the first stage, the goal is to reach a trap node-tuple. After that goal is achieved, then the goal of the second stage is to reach a violating node-tuple. There are two benefits of this two-stage search strategy. One is that the search in the second stage only needs to examine the trap node-tuples since a successor of a trap node-tuple is also a trap node-tuple. A second benefit is that there is probably no need to try to reduce the length of the path from the trap node-tuple to the violating node-tuple. This is based on the observation that if a counterexample contains trap node-tuples, analysts usually only need to study a prefix from the initial node-tuple to the first trap node-tuple, since this prefix is almost always enough to determine what went wrong in the verification. Thus, the requirements for these two stages are different. Stage 1 is required to quickly find a short trace to the trap node-tuple. Stage 2 is only required to find a violating node-tuple quickly, since the length restriction can be ignored.

Based on the different goals of these two stages, we can choose appropriate algorithms for each stage. For the first stage, we considered the BFS and WA_{TA} algorithms, since they tend to keep the paths short. We also considered BF_{trap} since it should help find a trap node-tuple quickly. In addition, we considered using a combination of the WA_{TA} and the BF_{trap} heuristics in which we first use BF_{trap} to select node-tuples, and then use WA_{TA} to break ties. This combination is denoted as $BF_{trap} + WA_{TA}$. We also tried using WA_{TA} first and then applying BF_{trap} to break ties. Experimentally, this second combination was just slightly better than applying WA_{TA} alone. Thus, we do not report further on this combination. Recalling that speed is the only requirement for the second stage, we considered DFS and BF_{TA}, since both tend to find a violating node-tuple fast.

4. METHODOLOGY

We considered four one-stage search algorithms: BFS, DFS, WA_{TA}, and BF_{TA}. For the two-stage heuristic search algorithms, we considered four alternatives for the first stage (BFS, WA_{TA}, BF_{trap}, and $BF_{trap} + WA_{TA}$) and two alternatives for the second stage (DFS and BF_{TA}), giving us eight two-stage algorithms. For algorithms using WA_{TA}, we considered a number of different values of the weight w, and report here the results for five of these (1, 2, 3, 5, and 9) that show the best results and the trend as w increases.

As mentioned before, two measurements, the runtime and the length of the counterexample, are used to evaluate the search algorithms. Runtime is used to measure the performance and is determined by a straightforward reading of the computation time. Length is used to evaluate the quality of the counterexample and is computed by counting the number of node-tuples in the counterexample. In the case where the counterexample has a trap node-tuple, however, we define the measure of length to be the length of the prefix of the counterexample that starts from the initial node-tuple and ends with the first trap node-tuple. We believe that it is this prefix that is usually of primary interest to the analyst trying to determine whether the counterexample represents a genuine violation of the property being checked and, when it does, to understand the source of the problem. All the properties in our experiment were in a form that had a trap state, so the lengths reported here are all lengths of such prefixes.

The runtimes and prefix lengths for the different algorithms vary considerably over the different subjects in our experiment. Rather than report the raw values of runtimes and prefix lengths, we report the ratios of the runtime and prefix length compared to the corresponding values obtained by running BFS on the same subject. Since BFS returns the shortest counterexample, we would expect the prefix length for the heuristics to be larger than the prefix length for BFS, but prefer algorithms where the ratio is closer to 1. (As described in the next section, the heuristic algorithms can sometimes do better than BFS because they are focusing on the trap node-tuple instead of the violating node-tuple.) Since BFS tends to take longer to execute than DFS, we would expect the heuristic algorithms to take less time to execute than BFS and thus prefer algorithms where the ratio is smaller.

We used a set of 10 different examples from the concurrency literature, including a memory management system, the gas station system, the three way handshake protocol, and the Chiron user interface system. These examples have been widely studied and are frequently used to compare the performance of FSV tools. Most of them are scalable, allowing an evaluation of performance as the size of the system being verified increases. For the scalable systems, we chose a size that was not too large for the search algorithms to find a counterexample with the resources available nor too small for differences in performance of the various algorithms to be detectable. We found that the pattern of performance of these algorithms on a scalable system did not change much with the size of the system, although the differences between algorithms were magnified as the size increased. We therefore chose one size for each of the scalable systems, one that was as large as possible without making the counterexample search run out of memory for most of the algorithms we considered. With these choices, the Ada programs used in the experiment had between 4 and 16 tasks and ranged in size from 226 to 6,935 lines of code, as shown in Table 1.

System	LOCs	# of tasks	# of properties	# of subjects
Memory management	703	9	1	4
Gas station	648	9	1	4
Three way handshake	491	5	1	4
Chiron(Original ver.)	6,495	8	3	12
Chiron(Decomposed ver.)	6,935	16	3	14
Token ring	386	11	2	24
Relay	321	7	1	9
Cyclic scheduler	226	11	1	9
Peterson n-way tie breaker	1,063	4	1	8
Smoker	327	7	6	14

Table 1: The example programs in the experiment

For each of these programs, we had already identified several properties and used FLAVERS to verify that each property holds. For this experiment, however, we needed properties that are violated (in the model, at least). Rather than introducing faults in the programs or properties that would cause the properties to be violated, we chose to simply delete some constraints that removed paths that violated the properties. While this may seem like it would lead to contrived examples, in fact it reflects a common situation in the use of FSV tools, in which the analyst tries to verify a property with a model that, though conservative, does not represent the system under analysis accurately enough. Analysts typically go through several cycles of running an FSV tool, finding a counterexample that does not correspond to an actual execution of the system, and refining the model to eliminate this sort of spurious counterexample (usually at the cost of making the model more complex and the verification more expensive).

For FLAVERS, this process of refining the model involves adding constraints. In previous work, we had identified a set of constraints for each system and property that would give a conclusive result and that was minimal, in the sense that no proper subset would give a conclusive result. Given such a minimal constraint set for a system-property pair, we generate a subject by removing one constraint from that set. As a result, if the minimal constraint set has order n, there will be n subjects for the system-property pair. Some of these subjects, however, had running times that were too small to be useful in this experiment. We discarded any subject for which any search algorithm took less than 2 seconds to find a counterexample. After removing these small subjects, we had 102 subjects from 20 different system-property pairs, as shown in Table 1. In all these cases, the property has a reachable trap state.

We ran the experiments on a PC with a 2 GHz Pentium 4 processor and 1 GB of memory running Linux. We collected the runtime information by using the Linux command "time." FLAVERS is written in Java and was run on Sun Java SDK Standard Edition (build 1.4.1_01).

There are several threats to the validity of our results. First, the selection of examples may bias the results. Most of our examples are relatively small, even with scaling, and represent somewhat unrealistic programs that have been constructed to illustrate issues in the design of concurrent systems. For each system, we verified a small number of properties. These examples may not adequately represent the range of systems and properties to which FLAVERS (or

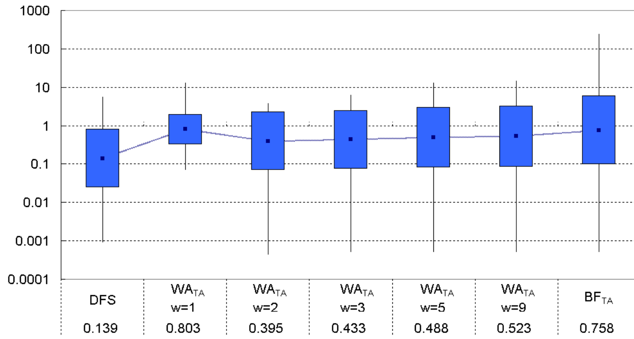


Figure 5: The average runtime ratios of one-stage algorithms

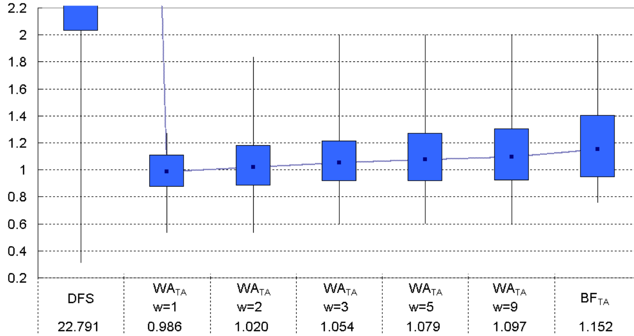


Figure 6: The average prefix length ratios of one-stage algorithms

other FSV tools) might be applied in practice, and our results may be misleading for that reason. A second threat arises from our method of generating subjects with counterexamples from subject for which the analysis was originally conclusive. As a consequence, the results reported in the paper may not reflect the performance of these algorithms in the cases where the property does not hold even when all necessary constraints are used. Finally, all properties used in the experiment have a reachable trap state. This means that we did not evaluate the performance of these algorithms in cases where the property does not have a reachable trap state. (Of course, our two-stage algorithms are not applicable for these cases.)

5. EXPERIMENTAL RESULTS

Here we present and analyze the experimental results. All the inputs and results from our experiment are available at `http://laser.cs.umass.edu/counterexamplesearch`.

Figure 5 shows the average runtime ratios (compared to BFS, as described earlier) for the one-stage algorithms. The vertical lines in the figure show the range of runtime ratios for each algorithm. The line running left to right through the boxes connects the geometric mean of the runtime ratios of the different algorithms;[4] the numeric values of these means are also given along the X axis below the names of the algorithms. The box gives the geometric standard deviation of each algorithm. Figure 6, similarly shows the average prefix length ratios for one-stage algorithms.

These two figures show that the DFS, whose average runtime ratio is 0.139, is the fastest among one-stage algorithms, but its average prefix length ratio, which is equal to 22.79, is the worst among them. For WA_{TA} and BF_{TA} (recall that the BF can be thought of as a special WA* with a very large weight), we observe that the average runtime ratios are all smaller than 1, meaning that on average these algorithms are faster than BFS, though not as fast as DFS. When the weight is set to 2, the WA_{TA} runs fastest. As the weight increases from 2, the average runtime ratios increase as well. The average prefix length ratios of WA_{TA} and BF_{TA} also increases with the weight, but all of them are around 1.

Figures 7 and 8 show the runtime ratios for the two-stage algorithms using DFS and BF_{TA} in the second stage, respectively. Figure 9 shows the prefix length ratios for two-stage algorithms. (Since we are only interested in the prefix

[4]Here we use the geometric mean of the ratios rather than the arithmetic mean. The reason is illustrated by the following simple example. Suppose there are 100 subjects and the runtime ratio of algorithm **A** for 50 subjects are all 0.01, meaning that algorithm **A** is 100 times faster than BFS for these subjects. For the remaining 50 subjects, suppose that the runtime ratios of **A** is 100, meaning that the algorithm A is 100 times slower than BFS. In this example, the arithmetic mean of the runtime ratios for **A** is around 50, suggesting that, on average, algorithm **A** is nearly 50 times faster than BFS even though **A** is 100 times slower than BFS on half the subjects. The geometric mean of the runtime ratios of **A**, however, is equal to 1, a more useful estimate.

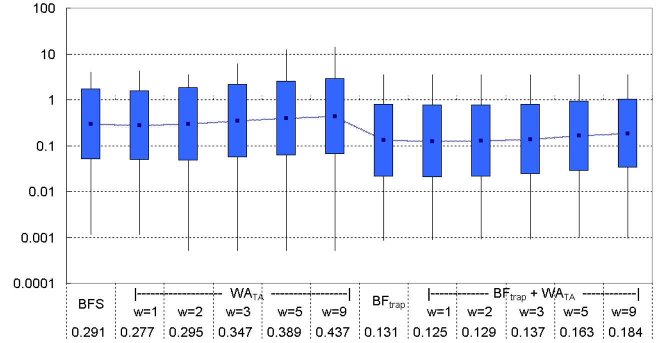


Figure 7: The average runtime ratios of two-stage algorithms that use the DFS in the stage 2

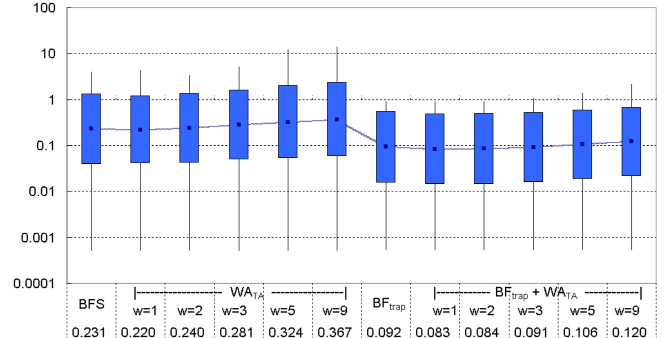


Figure 8: The average runtime ratios of two-stage algorithms that use the BF_{TA} in the stage 2

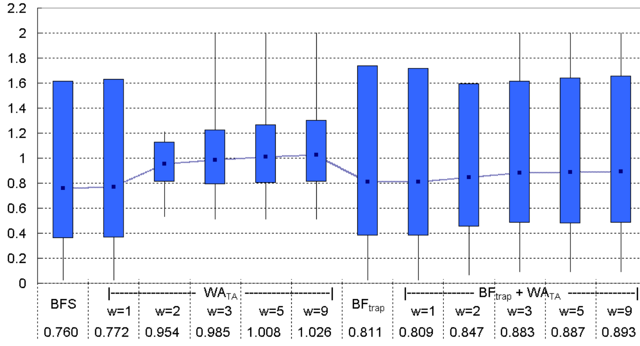


Figure 9: The average prefix length ratios of two-stage algorithms

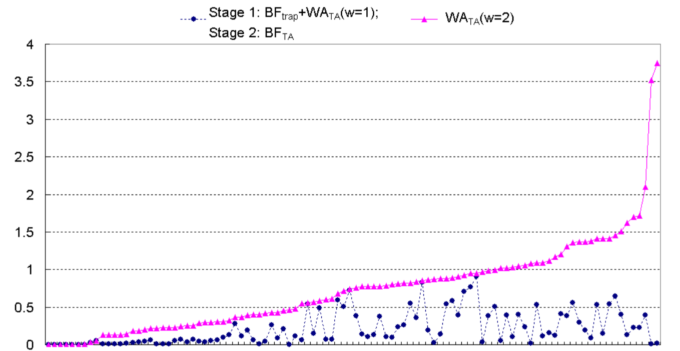


Figure 10: The runtime ratio comparison

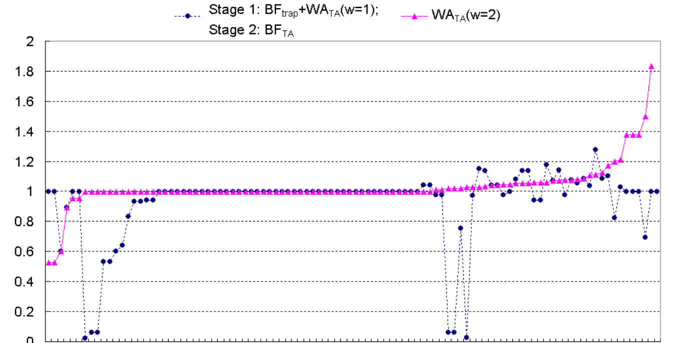


Figure 11: The prefix length ratio comparison

length, it is not necessary to distinguish between the different possibilities for the second stage.) All these figures use the same format as Figures 5 and 6.

Figure 7 and Figure 8 show that the speed of the algorithms using BFS for stage one is relatively close to that of the algorithms using WA_{TA}, compared to the difference between these when used alone as one-stage algorithms. Here WA_{TA} performs best with a weight of 1, and the performance decreases as the weight increases. The algorithms that use the trap heuristic in stage one are faster than those that do not. Using WA_{TA} after the BF_{trap}, as compared to using the BF_{trap} only, improves the search speed when the weight is small, but hurts performance when the weight is larger than 3.

Finally, comparing Figure 7 with Figure 8, we can see that the BF_{TA} is faster than the DFS in stage 2 for these examples.

Figure 9 shows that all two-stage algorithms have small average prefix length ratios, mostly less than 1. This may be counterintuitive, but the BFS, when used as a one-stage algorithm, considers the length of the whole counterexample. As a result, the BFS always finds the shortest counterexamples, but these counterexamples may not have the shortest prefixes from the initial node-tuple to a trap node-tuple. Since what we are evaluating here is the length of this prefix, rather than the length of the whole counterexample, the average prefix length ratios of these two-stage algorithms may be, and often are, smaller than 1. When the BFS is used as an alternative in stage 1, it does find the counterexample with the shortest prefix, as shown in Figure 9, where the average prefix length ratio for BFS, 0.760, is the smallest.

Among these two-stage algorithms, the one that uses BF_{trap} + WA_{TA} with $w = 1$ in stage 1, and uses the BF_{TA} in stage 2 seems to offer a good combination of speed and counterexample quality. Its average runtime ratio is 0.083, which means that this algorithm is on average about 12 times faster than the BFS, and is even faster than the DFS, whose average runtime ratio is 0.139. Meanwhile, the average prefix length ratio of this two-stage algorithm is 0.809, which is only slightly larger than 0.760, the shortest prefix length ratio of all.

To give a more concrete idea about how this two-stage algorithm compares with the WA_{TA} with $w = 2$ (which offered a good combination of speed and counterexample quality among the one-stage algorithms), we show the runtime ratios and prefix length ratios comparisons case by case between these two algorithms in Figure 10 and 11. Figure 10 shows that the two-stage algorithm runs faster than the one-stage algorithm for every subject. Moreover, all runtime ratios of the two-stage algorithm are less than 1, whereas more than 1/4 of runtime ratios of the one-stage algorithm are larger than 1. From Figure 11, we see that most of the prefix length ratios for the two-stage algorithm are less than or equal to those of the one-stage algorithm.

Thus, these experimental results show that the BF_{trap} + WA_{TA} algorithm with $w = 1$ in stage 1 BF_{TA} in stage 2 algorithm is relatively effective at producing a short counter example quickly.

6. RELATED WORK

The TA heuristic was first presented in [5] and used there in the A* algorithm. We extend that work by applying the TA heuristic in WA* and BF algorithms and by developing the trap heuristic and a two-stage algorithm based on the trap state in the property. The focus of the earlier work was on comparing different algorithms used in different situations, namely when the property to be verified is expected to hold or not. That work noted that although it would be desirable to have one algorithm that would work well in both situations, different algorithms were needed to achieve good performance. As shown in [5], for the case where the property does not hold, the A* algorithm using the TA heuristic is preferred; but for the case where the property holds, a State Propagation algorithm gives the best performance. As

described here, we are able to develop a heuristic search algorithm that performs better than the A* algorithm with the TA heuristic for the situation where the property does not hold. We also compared our best heuristic search algorithm with the best State Propagation algorithm for the situation where the property holds. In this situation, all node-tuples reachable from the initial node-tuple must be examined, no matter which algorithm is used. In this case, our heuristic algorithm is about 30% slower than the State Propagation algorithm, since it must compute evaluation functions and sort the worklist based on these values. Hence, when analysts expect that the property holds, the State Propagation algorithm should still be used. When the property is not expected to hold, such as during the early stages of the development, however, our heuristic search algorithm would be the appropriate algorithm to use.

A number of other investigators have applied heuristic-guided search to find counterexamples with FSV tools. In [10], for example, heuristics based on the property being verified are proposed for use with a version of SPIN. SPIN uses a nested depth-first search algorithm to find an accepting state in a cycle; the work in [10] uses the structure of the property to eliminate the need for the "inner" part of the search in some cases and uses heuristics to guide the "outer" part of the search. In contrast, our two-stage algorithm employs heuristics in both stages and uses the structure of the property to determine if we can initiate the second stage to improve efficiency. In [13], a heuristic considering code coverage is used with Java PathFinder. This heuristic, in essence, is based on the structure of the model. The TA heuristic in our work is based on some aspects of the model's structure. A Hamming Distance heuristic has been used to guide the counterexample search in both Murϕ [26] and SPIN [9]. These techniques require that a specific state of the system in which the property has been violated be identified initially; the Hamming distance between that state and the system state currently being expanded is then used to guide the search for a short counterexample that reaches the given violation state. Our two-stage heuristics do not search for a path to a specific state of the system, but begin by searching for a path to any node-tuple in which the property is in the trap state. In [11], a genetic algorithm is used to help find error states in very large systems. This approach makes use of several heuristics such as counting the number of outgoing transitions, maximizing assertion evaluation, and maximizing exchanged messages to find deadlocks and violations of assertions, but this work is focused on finding an error state and does not consider the length of the path to that state.

There have been many research efforts aimed at addressing the computational complexity of search by dividing the entire process into multiple phases. One example is the use of iterative improvement (or iterative repair) techniques in planning and scheduling (e.g., [22, 23]). These techniques typically divide the search into two phases. In the first phase, a solution candidate is generated using randomized techniques or some fast greedy heuristic algorithm. In the second phase, the solution is improved using local search techniques until a local optimum is reached. Another example is the wide use of hybrid search in solving constraint satisfaction and combinatorial optimization problems (e.g., [14, 24]). These techniques typically combine some form of complete search with local search algorithms. The complete search, normally in the form of constraint propagation, is used to restrict the neighborhood or to prune the search space, while local search helps reach solutions quickly. Similar hybrid search techniques have been used to improve the efficiency of evolutionary programming techniques [25]. There has also been work on the application of hybrid search to heterogeneous search spaces. For example, the mixed-integer programming technique is an operations research paradigm that combines linear programming and branch-and-bound search [12]. What is unique about our problem domain and solution technique is that the two phases of the two-stage search strategy have different objectives to optimize, while sharing the same underlying representation and search graph. The two phases of the search are designed to reduce the overall computation time, while optimizing solution length in the first phase of the search.

7. CONCLUSION

FSV techniques can prove that a property holds on all possible executions of a system. Such methods are especially important for concurrent and distributed systems, where nondeterministic behavior makes testing especially problematic. FSV tools work by constructing a finite model that represents all the executions and then by verifying that the property holds on that model. When the property does not hold for the model, either because the model overapproximates the set of possible executions or because the system does not satisfy the property (or both), these tools produce a counterexample, a trace through the model that shows how the property can be violated. These counterexamples are of substantial value to analysts in refining the model as necessary and correcting actual faults in the system. But counterexamples that are long are hard to understand, and their utility is significantly reduced. It is therefore desirable that FSV tools return results quickly and, if the property does not hold, return short counterexamples.

For many FSV tools, the verification process can be regarded as a search over a graph representation of the model of the system. Standard search strategies, such as DFS and BFS, are usually not effective in satisfying both the time and length requirements. In this paper, we have explored heuristic-guided search strategies that do a better job in satisfying these requirements together. We have presented two heuristics, one based on the structure of the model and one based on the structure of the property being checked, and a novel two-stage search strategy. The two-stage strategy seeks to optimize different functions in the different stages, considering length in the first stage and speed in the second, although both share the same underlying representation and search graph. We have compared several search algorithms employing the heuristics, along with several different versions of the two-stage strategy that use different heuristics to guide the search during the different stages. One of these versions, employing $BF_{trap} + WA_{TA}$ with $w = 1$ in the first stage and BF_{TA} in the second stage, seems to provide a particularly good combination of speed and short counterexamples.

We intend to explore several additional directions. First, we will see how well our methods work on a broader range of systems and properties. While we will certainly investigate the use of these search strategies on additional Ada programs, the FLAVERS approach can be applied to Java programs as well as Ada programs [20, 21], and we are currently building tools that use the Bandera toolset [7] to con-

struct FLAVERS models from Java code. As soon as these tools are complete, we will investigate the applicability of these heuristics and the two-stage strategy to counterexample search in the analysis of Java programs. It will be interesting to see if the experimental results are consistent for these two languages.

We are also interested in exploring the use of heuristic search in automated counterexample guided refinement [2–4, 17]. In this work, a counterexample that reflects the model's overapproximation of the system's execution is used to automatically refine the model, and alternating cycles of counterexample search and refinement continue until the FSV tool runs out of resources or can give a definitive statement as to whether the property holds for the system. (In FLAVERS, the model is refined by adding additional constraints.) Several techniques have been proposed for analyzing counterexamples to determine how to refine the model and, since these techniques are automated, the length of the counterexample may be less significant than when the counterexample will be viewed by a human analyst. Other aspects of the counterexample such as the prefix to a trap node-tuple, however, may lead to more rapid refinement of the model, and we plan to investigate the extent to which heuristic search can improve this process.

8. ACKNOWLEDGMENTS

We thank Jamieson Cobleigh and Heather Conboy for their assistance in collecting the experimental subjects.

9. REFERENCES

[1] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2(3):117–126, 1987.

[2] T. Ball, A. Podelski, and S. K. Rajamani. Relative completeness of abstraction refinement for software model checking. In *8th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 158 – 172. Springer-Verlag, Apr. 2002.

[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*, volume 2057 of *LNCS*, pages 103–122. Springer-Verlag, May 2001.

[4] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proceedings of the 12th International Conference on Computer Aided Verification*, volume 1855 of *LNCS*, pages 154–169. Springer-Verlag, Jul. 2000.

[5] 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, May 2001.

[6] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil. FLAVERS: A finite state verification technique for software systems. *IBM Systems Journal*, 41(1):140–165, 2002.

[7] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the 22nd International Conference on Software Engineering*, pages 439–448, Jun. 2000.

[8] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[9] S. Edelkamp, A. L. Lafuente, and S. Leue. Trail-directed model checking. In *Proceedings of the Workshop on Software Model Checking, Electrical Notes in Theoretical Computer Science*, volume 55, Jul. 2001.

[10] S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology Transfer*, 5(2-3):247–267, 2004.

[11] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *8th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, volume 2280 of *LNCS*, pages 266–280. Springer-Verlag, Apr. 2002.

[12] C. Gomes and B. Selman. Hybrid search strategies for heterogeneous search spaces. *International Journal on Artificial Intelligence Tools*, 9(1):45–57, 2000.

[13] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 12–21, Jul. 2002.

[14] D. Habet, C. M. Li, L. Devendeville, and M. Vasquez. A hybrid approach for SAT. In *Proceedings of the 8th International Conference on Principle and Practice of Constraint Programming*, pages 172–184, Sep. 2002.

[15] E. A. Hansen, S. Zilberstein, and V. A. Danilchenko. Anytime heuristic search: First results. TR UM-CS-1997-50, Department of Computer Science, University of Massachusetts Amherst, 1997.

[16] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[17] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 58–70, Jan. 2002.

[18] G. J. Holzmann. *The* SPIN *Model Checker*. Addison-Wesley, Boston, 2004.

[19] G. Naumovich and G. S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the 6th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34, Nov. 1998.

[20] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the 21st International Conference on Software Engineering*, pages 399–410, May 1999.

[21] G. Naumovich, G. S. Avrunin, and L. A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 338–354. Springer-Verlag, Sep. 1999.

[22] G. Rabideau, S. Chien, J. Willis, and T. Mann. Using iterative repair to automate planning and scheduling of shuttle payload operations. In *Proceedings of the 11th Conference on Innovative Applications of Artificial Intelligence*, pages 813–820, 1999.

[23] D. Ratner and I. Pohl. Joint and LPA*: Combination of approximation and search. In *Proceedings of the 5th National Conference on Artificial Intelligence*, pages 173–177, 1986.

[24] P. Shaw. Using constraint programming and local search methods to solve vehicle routing problems. In *Proceedings of the 4th International Conference on Principles and Practice of Constraint Programming*, pages 417–431, 1998.

[25] V. Tam and P. Stuckey. Improving evolutionary algorithms for efficient constraint satisfaction. *International Journal on Artificial Intelligence Tools*, 8(4):363–383, 1999.

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