Heuristic-Based Model Refinement for FLAVERS

Jianbin Tan, George S. Avrunin, and Lori A. Clarke
Laboratory of Advanced Software Engineering Research
Department of Computer Science
University of Massachusetts
Amherst, Massachusetts 01003, USA
{tjb, avrunin, clarke}@cs.umass.edu

Abstract

FLAVERS is a finite-state verification approach that allows an analyst to incrementally add constraints to improve the precision of the model of the system being analyzed. Except for trivial systems, however, it is impractical to compute which constraints should be selected to produce precise results for the least cost. Thus, constraint selection has been a manual task, guided by the intuition of the analyst. In this paper, we investigate several heuristics for selecting task automaton constraints, a kind of constraint that tends to reduce infeasible task interactions. We describe an experiment showing that one of these heuristics is extremely effective at improving the precision of the analysis results without significantly degrading performance.

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. They do suffer, however, from the so-called state-explosion problem, where the size of the model or the analysis costs may be exponentially large with respect to the size of the system being analyzed.

There are a number of different techniques for dealing with this state-explosion problem. Many of these expect the analyst to have sufficient insight to be able to create an abstract model of the system that can serve as the basis for efficient verification. Usually the first few abstract models that an analyst creates are too large, and the analyst must think of additional abstractions that will eliminate some of the information in the model while maintaining the soundness of the proof process. After several attempts, analysts can often create a model that is sufficiently small for either proving the property or revealing a counterexample trace that exposes how the system violates the property.

FLAVERS [6, 9], FLow Analysis for Verifying Systems, uses data-flow analysis techniques to reason about properties described in terms of sequences of events. Using program analysis techniques, FLAVERS automatically creates a compact, but imprecise, representation of the system that can then be augmented with *constraints* that add information to improve the precision of the analysis results. Thus, instead of trying to develop abstractions to help *reduce* the size of the model, with FLAVERS analysts try to define constraints that judiciously *increase* the size of the model.

Constraints are a very general mechanism that may be used to model the values of variables, restrict the flow of control, or constrain the inputs from the execution environment. When FLAVERS returns a counterexample trace, this trace shows where the model appears to violate the property. Analysts can then usually determine if the trace represents a true violation or, when the counterexample is spurious, select some additional constraints to increase the precision of the model and eliminate this spurious counterexample. Experimental studies have shown that this approach is very effective [10], and a comparison of FLAVERS with other finite-state verification tools [1] shows that FLAVERS usually performs as well as, and often better than, such tools as SPIN [17] and SMV [19].

In selecting constraints, the analyst is making a trade-off between performance and precision. Additional constraints usually improve the precision of the results, but increase the analysis costs. We know of no way to determine which constraints should be selected to produce precise results at the least cost without actually carrying out the analysis with all possible sets of constraints. Constraint selection, therefore, has largely been a manual process, guided by the intuition of the analyst. In this paper, however, we present heuristics for automatically selecting task automaton constraints, a kind of constraint that removes some infeasible paths from the model. We propose and evaluate four heuristics and demonstrate that one of these is particularly effective at predicting the task automaton constraints that should be included. The

evaluation is conducted using the FLAVERS/Ada toolset applied to a set of Ada tasking programs.

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 each of these heuristics and is followed by a section describing our experimental methodology. Section 5 presents our experimental results, and Section 6 describes related work. In the conclusion, we summarize our results and describe directions for future work.

2 FLAVERS Overview

FLAVERS/Ada is a finite-state verification tool that uses data-flow analysis techniques to verify that all possible executions 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 specification, involving events **e1** and **T2.synch**, for the system of three communicating tasks described in Figure 1(b). This property specifies that event **e1** always occurs, but only after task **T2** has synchronized with at least one of its client tasks, and task **T2** synchronizes with its clients twice.

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 model 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) [20] 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 $L : N \rightarrow \mathcal{A}_G \cup \{\emptyset\}$ 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 circular nodes represent local nodes, the diamond-shaped nodes represent communication nodes, which model the Ada rendezvous, and the triangular nodes represent the initial and final nodes. Note that a local node is associated with only one task, while a communication node represents the communication between two participating tasks, and the initial and final nodes of the TFG represent the initial and final nodes of each task. Thus, the control flow graph for each task is a subgraph of the TFG. We use $N_l(t)$ to denote the set of all local nodes of task t, and $N_c(t)$ to denote the set of all communication nodes of task t. For instance, in figure 1(d), $N_l(\textbf{T1}) = \{1, 6\}$ and $N_c(\textbf{T1}) = \{4\}$. And we define $N(t) = N_l(t) \cup N_c(t)$.

The solid edges in Figure 1(d) represent control flow within a task and are called *local edges*. The dashed edges represent MIP edges. For concurrent systems, the number of MIP edges may be very large. Partial order reduction [23] and other optimization techniques are used to eliminate unnecessary MIP edges. After such optimizations are applied to the example, only two MIP edges remain, as shown in Figure 1(d).

The 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 model that results in traversing the same sequence of events. Therefore, when FLAVERS does not find a path in the model that can violate the property, the analyst can be sure that the property holds on the original system.

Given the TFG and the property, FLAVERS uses a fixed-point algorithm, called *state propagation* [9, 24] to determine what states in the property should be associated with each node in the TFG. Each path through the TFG determines a sequence of events. A state s is associated with a node n if, for some path from $n_{initial}$ to n, the corresponding sequence of events drives the property to state s. The algorithm is a typical forward-flow, any-path data-flow analysis problem [18]. To determine whether the property holds, FLAVERS checks the set of states associated with the final node. For a desirable property, if there are only accepting states in the set, FLAVERS returns *conclusive*, meaning that

[1] FLAVERS currently does not handle recursive calls.

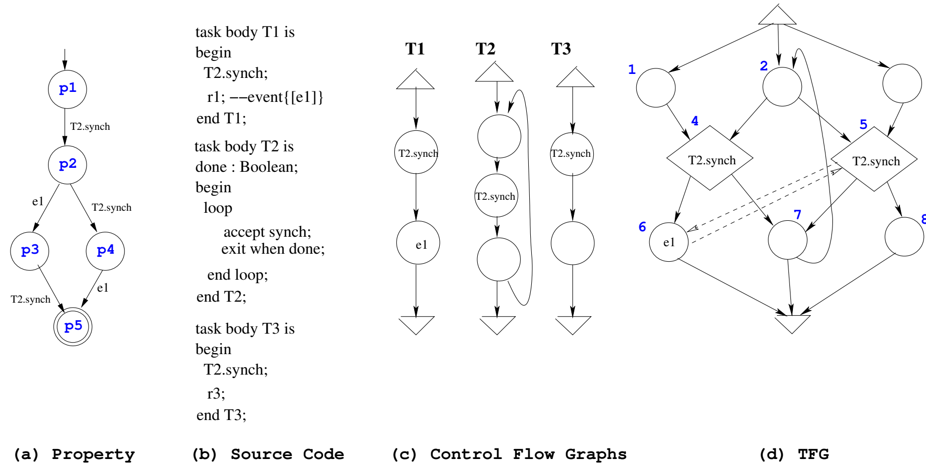

Figure 1. A simple example

the property holds. Otherwise, FLAVERS returns *inconclusive* and provides a *counterexample* path through the model to show how the property may be violated.

The model created by FLAVERS is compact and conservative, but is imprecise since it over-approximates the event sequences allowed by the system. This means that some event sequences that appear in the model do not arise on any actual execution through the system. Thus, when FLAVERS returns an inconclusive finding and a counterexample path that violates the property, it may be an indication of an error in the system (or in the property), or it may be that the corresponding path in the system is infeasible. If all the possible counterexample paths are infeasible, then the inconclusive finding is a *spurious* false finding and the property does hold.

In the example, the path $n_{initial}, 2, 4, 7, 2, 5, 7, n_{final}$ in Figure 1(d) leads the property in Figure 1(a) to non-accepting state p4, and thus the path violates the property. This path is not feasible since, for instance, the statement in task **T1** corresponding to node 6 must be executed before task **T1** terminates.

One of the strengths of FLAVERS is that an analyst 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. The state propagation algorithm simultaneously incorporates the property and all of the constraints into the analysis by associating sets of tuples with each node in the TFG, where the nth element of the tuple corresponds to the state of the nth FSA.

In FLAVERS, there are three kinds of commonly used constraints: *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 a task. CAs are usually based upon external knowledge about the environment in which the system will be executed, and thus it would be difficult to automatically predict this information (although there has been work on defining the weakest such environment [25, 27]). In our experience, the analyst often has a good sense of the VAs that are likely to impact the analysis and therefore does a reasonable job manually selecting VAs for inclusion. Unfortunately, this does not seem to be the case for TAs. Thus, as our first step toward automatically selecting constraints, we focus our work on selecting TAs and assume that VAs and CAs will be selected by the analyst.

Paths through the TFG cross between different tasks through communication nodes, which represent points where the tasks synchronize, and through MIP edges, which represent the different interleavings of events from different tasks. The idea behind the TA is that if a TFG path is feasible, the projection of the path on every task should also be a feasible path through the task's control flow graph. As an example, consider the infeasible path we mentioned above. If we project this path on task **T1**, we get the sequence $n_{initial}, 4, n_{final}$. The projection is obviously infeasible, since node 6 is skipped. A TA can be automatically created for any task so that whenever its flow of control is violated, the appropriate tuple can be discarded from further consideration during state propagation since this represents non-executable control flow.

TAs are thus used to eliminate from consideration certain paths through the TFG. These paths exist because of the representation of communication and interleaving in the TFG, but do not adhere to the flow of control in the individual tasks. It can be extremely difficult for the analyst to anticipate exactly which such paths may need to be eliminated—most of the errors in programming concurrent systems are probably due to interleavings of execution that were not

foreseen by the developers—and this is what makes it difficult for the analyst to know which TAs to select. Although TAs can be automatically generated for each task in a system, the cost of the analysis usually increases with the number of constraints that are included. Thus, we need a strategy for helping analysts select the TAs that should be included.

If insufficient or inappropriate constraints are selected, FLAVERS will return spurious false findings, whereas selecting too many constraints will usually significantly increase the time and space requirements. Table 1 illustrates how the choice of TAs affects the analysis result and the analysis performance for a small memory management system. In Table 1 there are three selected TA sets. Set **A** uses only two TAs, which are not sufficient to prove the property, and thus FLAVERS returns an inconclusive result with this set. Set **B** and set **C** have six and eight TAs respectively, and with either set FLAVERS returns a conclusive result. With set **C**, however, FLAVERS uses over seven times as much time and space[2] as with set **B**. This example demonstrates the importance of selecting a good set of TAs.

Selected TA set	**A**	**B**	**C**
Size of the set	2	6	8
Result	Inconclusive	Conclusive	Conclusive
Runtime	1.47s	9.71s	68.33s
Space	100,704	5,188,736	37,056,080

Table 1. An example showing how the selection of TAs affects the result and performance

Currently, FLAVERS does not automatically include any TAs nor provide any guidance on which TAs might be candidates for consideration. Usually an analyst needs to study several counterexample paths and experiment with adding and removing TAs before finally arriving at a set that efficiently leads to either a conclusive result or a helpful counterexample path that exposes a fault in the system. It would save the analyst considerable effort if this process could be at least partially automated by employing heuristics.

3 Heuristics

It is natural to base heuristics for selecting TAs on information from the system, the property, and the other selected constraints. Intuitively, if an event that labels a node in a TA (and hence also a node in the TFG, since the labels for TA nodes are obtained by regarding the TAs as subgraphs of the TFG) also appears in the property or in a VA or CA, then that TA may affect the verification and thus should be selected. It is this intuition that guides our exploration of heuristics.

We refer to the input to a TA selection heuristic, including the system, the property, and the initially chosen VAs and CAs, as a *subject*. The output is a set of selected TAs, denoted by C_{PT}. We use $\mathcal{A}$ to denote the set of all events mentioned in the property, VAs, and CAs in a subject, and refer to $\mathcal{A}$ as *the subject alphabet*. All our heuristics are based on the intuition described above and thus rely on this subject alphabet.

Heuristic $H(\mathcal{A})$**:** This heuristic selects any TA with a node labeled by an event in the subject alphabet. Formally, for task t, if there is a node $n \in N(t)$ such that $L(n) \in \mathcal{A}$, then we put t in C_{PT}. For example, when this heuristic is applied to the subject described in Figure 1, we get $C_{PT} = \{$**T1, T2, T3**$\}$. (Note that there is no VA or CA in this subject).

The experimental results described later show that this heuristic tends to select too many TAs. The main reason is that communication nodes always belong to two tasks. Once a communication node is labeled by an event in the subject alphabet, both TAs will be selected. To overcome this shortcoming, we introduce heuristics that treat local nodes and communication nodes separately.

Heuristic $H(\mathcal{A}_l)$**:** This heuristic chooses TAs based on local nodes only. The idea is that if the task has any local node labeled by an event in the subject alphabet, then that TA should be selected. Formally, for task t, if there is a node $n \in N_l(t)$ such that $L(n) \in \mathcal{A}$, then we put t in C_{PT}. For example, when this heuristic is applied to the subject in Figure 1, we get $C_{PT} = \{$**T1**$\}$.

Heuristic $H(\mathcal{A}_c)$**:** Based on our observation that a communication node is always included in two tasks but usually selecting only one TA will be enough, we developed this heuristic with two principles in mind. First, all communication nodes whose labeled events are in $\mathcal{A}$, called *communication alphabet nodes*, must be covered by at least one selected TA, where a communication node is *covered* by a TA if the node belongs to that TA and the TA is selected. Second, as few communication alphabet nodes as possible should belong to more than one selected TA.

The algorithm repeatedly selects the TA with the most uncovered communication alphabet nodes until all the communication alphabet nodes are covered. At any point, of course, there may be more than one task with the same maximal number of uncovered communication alphabet nodes, and we need a procedure to break the tie. (To keep the number of selected TAs small, we do not want to routinely select all these TAs.) As noted earlier, many of the infeasible paths through the TFG that need to be eliminated by TAs

[2] Space is computed by summing the sizes of all tuple sets associated with TFG nodes after the state propagation algorithm terminates. This is an approximate evaluation of memory usage.

```
Heuristic H(A_c) {
  Set C_PT = ∅;
  Loop forever {
    //Find covered communication alphabet nodes
    N_cov := ⋃_{t∈C_PT} {n|n ∈ N_c(t) ∧ L(n) ∈ A};
    For each task t ∉ C_PT
      N_uncov(t) := { n ∈ N_c(t) |  L(n) ∈ A ∧ n ∉ N_cov };
    Let maxUncov = max_{t∉C_PT} |N_uncov(t)|;
    If maxUncov == 0, then return C_PT;
    else {
      Add all tasks t with N_uncov(t) == maxUncov to set S;
      Let maxMipDen = max_{t∈S} mipDensity(t);
      Add all tasks t ∈ S with mipDensity(t) == maxMipDen to C_PT;
      }
    }
  }
// Compute normalized number of MIP edges of task t
Function mipDensity(t) {
  //Determine the MIP edge set of task t
  Let M(t) = {e = (m,n)|m ∈ N(t) ∨ n ∈ N(t)};
  Let n_TA(t) be the number of nodes in the TA of task t;
  return |M(t)| ÷ n_TA(t);
}
```

Figure 2. The algorithm for heuristic $H(\mathcal{A}_c)$

arise from the MIP edges representing the interleaving of the execution of different tasks. Large TAs (regarded as subgraphs of the TFG) are likely to have more MIP edges, but also to make the analysis more expensive. We therefore use the number of MIP edges entering or leaving the task, normalized by the number of nodes in the TA, to select the TA. If there is still a tie, we take all the TAs with the highest normalized number of MIP edges.[3] Figure 2 shows the algorithm used. When this heuristic is applied to the subject in Figure 1, we get $C_{PT} = \{\mathbf{T2}\}$.

Heuristic $H(\mathcal{A}_l) + H(\mathcal{A}_c)$: By combining $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$ we can take into account all the events in the subject alphabet without necessarily selecting all the TAs chosen by $H(\mathcal{A})$. We note that the output of heuristic $H(\mathcal{A}_c)$ can be affected by the initial value of C_{PT}. This means that there are two different ways to combine $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$. We could simply apply the two heuristics independently and take the the union of the results returned. Or we could apply $H(\mathcal{A}_l)$ first and use the TAs selected by it as the initial value of C_{PT} when applying $H(\mathcal{A}_c)$. For example, when applying the two heuristics independently to the subject of Figure 1 and taking the union of the results, we get $C_{PT} = \{\mathbf{T1}, \mathbf{T2}\}$. Applying $H(\mathcal{A}_l)$ first and then applying $H(\mathcal{A}_c)$, however, gives $C_{PT} = \{\mathbf{T1}, \mathbf{T3}\}$. In our experiments, these two ways of combining $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$ performed very similarly, but there was a slight advantage to applying $H(\mathcal{A}_l)$ first. For simplicity, in this paper we only report the results for the latter combination, which we refer to as $H(\mathcal{A}_l) + H(\mathcal{A}_c)$.

[3] We investigated a number of variants of this algorithm and found that the algorithm presented here works slightly better.

4 Methodology

In this section, we describe the methodology used in evaluating the four heuristics.

Example Systems: We used a set of 20 different examples from the concurrency literature. These included several versions of the dining philosophers, a memory management system, the gas station, some communication protocols, and the Chiron user interface system. These examples have been widely studied and are frequently used to compare the performance of finite-state verification tools. Most of them are scalable, allowing an evaluation of performance as the size of the system being verified increases. These Ada programs had between 3 and 22 tasks and ranged in size from 44 to 2,734 lines of code.

For each system, we selected from 1 to 10 properties, all of which hold for that system. There are several reasons why we only considered properties that hold. If a property is not satisfied by the system, FLAVERS will always report an inconclusive result, no matter which TAs are selected. Under these circimstances it would be difficult to evaluate the impact of TA selection on the analysis. Although the set of TAs used in the analysis may affect the quality of the counterexamples produced (in terms of their feasibility, understandability, etc.), evaluating this quality is complex. Moreover, as shown in [5], it may be advantageous to modify the FLAVERS algorithm in cases where inconclusive results are expected, and then additional heuristics to guide the search for counterexamples are likely to be helpful [12]. In future work, we intend to examine the use of heuristics when properties are violated.

Combining the different sizes of the scalable systems and the multiple properties, we had a total of 249 system-property pairs. For each of these, we chose a set of VAs and CAs that, together with some or all of the TAs, is sufficient for FLAVERS to show conclusively that the property holds. The VAs and CAs that we used were ones that were previously selected by analysts for these properties. All of the subjects and the results of our experiments are available at `http://laser.cs.umass/taselection`.

Measures: If we were only interested in conclusive results, we could simply select all the TAs. But we are trying to find ways to select TAs that will lead to conclusive results and provide good performance. We therefore need to find appropriate measures of the performance of our heuristics.

We selected three basic measures and evaluated each subject, with each heuristic, for each of these measures. To

measure precision, we counted the number of subjects for which analysis with the TAs selected by that heuristic gave a conclusive result. To measure performance, we considered both time and space. Time was determined by a straightforward reading of the computation time. Space was measured by the number of node-tuple pairs generated in the analysis.

Using these measures, we can compare the heuristics against each other. But we are really interested in comparing the heuristics against the best possible set of TAs. To do so, we would need to find the set of TAs that leads to the fastest analysis or the smallest number of node-tuple pairs while still producing a conclusive result. Unfortunately, finding such a set is impractical for all but the smallest examples. Adding TAs usually increases the time and memory required, but it does not always do so. Therefore, finding the optimal set of TAs would require running FLAVERS with each subset of the set of TAs for that system. For a subject with 20 tasks, this would require 2^{20} analysis runs.

We can, however, use the rule of thumb that small sets of TAs that lead to conclusive results give the best performance, although it can sometimes happen that adding TAs to a set that already gives a conclusive result will improve performance. In previous work, we had identified a minimal TA set giving a conclusive result for each subject. By *minimal*, we mean here that no proper subset of that set of TAs would lead to a conclusive result. Such a set need not be a *minimum*, in the sense that no other set with fewer TAs would produce a conclusive result.[4] We therefore use our known minimal set as a rough indicator of optimal performance. We can then evaluate the heuristics by comparing the time and space used with the TAs selected by the heuristics with the corresponding figures using this minimal TA set.

Because it is usually true that including more TAs than necessary to achieve a conclusive result leads to worse performance, we also compare the set of TAs selected by our heuristics with the minimal set. In particular, we report on how often the selected set is a proper superset or subset of the minimal set and how often the selected set is equal to the minimal set. Much more sophisticated measures of the difference between the selected TA set and the minimal set are certainly possible. Given that our minimal sets are not necessarily the sets producing the best possible performance, however, we do not think that such sophisticated measures are warranted.

Procedure: 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 Linux command "time." We ran each analysis problem three times and computed the average runtime. All the programs are written in Java and were run on Sun Java SDK Standard Edition (build 1.4.1_01).

Threats to Validity: 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 other finite-state verification tools) might be applied in practice, and our results may be misleading for that reason.

A second threat arises from our selection of a fixed set of VAs and CAs for each example. The performance of FLAVERS is certainly affected by the choice of VAs and CAs. While we have discussed our reasons for focusing here on the selection of TAs and noted that the choice of VAs and CAs is usually much more obvious, we have not carefully investigated the interaction between our heuristics and the particular selection of VAs and CAs we used.

Finally, we note again that the properties we chose are all satisfied by the systems we studied. The problem of selecting TAs, and guiding the search for counterexamples, when the property is not expected to hold is somewhat different, and it may be that our results do not hold in such cases.

5 Experimental Results

In this section we present the results from our experiment in terms of precision, time, and space.

For each heuristic, Table 2 shows the number of subjects for which the verification returned conclusive, inconclusive, or out-of-memory results. It also shows the conclusive percentage; that is, the number of subjects for which conclusive results were returned divided by the total number of subjects in the experiment. From this table we see that heuristics $H(\mathcal{A})$ and $H(\mathcal{A}_l) + H(\mathcal{A}_c)$ are effective, in the sense that more than 80% of the subjects get conclusive results. Heuristics $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$, however, are only effective for about 50% of the subjects. Thus it appears that both local and communication nodes should be considered when selecting TAs.

Table 2 also shows that four subjects get out-of-memory results with heuristic $H(\mathcal{A})$. Looking at these subjects, we found that the selected TA sets are much larger than the corresponding minimal TA sets. There is also one subject that gets an out-of-memory result with the TAs selected by $H(\mathcal{A}_c)$. For this subject, this heuristic does not select any TAs whereas the corresponding minimal TA set has four

[4] Finding a minimum set of TAs is an NP-complete problem [26]. In a few small cases, we do know that the minimal set we use is the unique minimal set, so that it is a minimum and a conclusive result will only be obtained if the selected TAs include all the elements of the minimal set.

TAs. This illustrates the atypical, but possible, situation noted above where adding TAs improves performance.

	$H(\mathcal{A})$	$H(\mathcal{A}_l)$	$H(\mathcal{A}_c)$	$H(\mathcal{A}_l) + H(\mathcal{A}_c)$
Conclusive	214	132	114	204
Inconclusive	31	117	134	45
Out-of-Memory	4	0	1	0
Total	249	249	249	249
Conclusive Percentage	85.9%	53.0%	45.8%	81.9%

Table 2. Result comparisons

Conclusive and inconclusive results are a crude measure of effectiveness, since we know that adding constraints tends to improve precision. Thus, we need to compare how close the selected sets are to the minimal TA sets we had identified in prior work. Table 3 shows the number of subjects for which the TA set selected by each heuristic is in the listed relationship with the minimal TA set. We see that $H(\mathcal{A}_l)+H(\mathcal{A}_c)$ selected exactly the TAs in the minimal set for 156 subjects and had only 48 superset and 31 subset results. Although heuristic $H(\mathcal{A})$, has the highest conclusive percentage, it gets 151 superset results and 67 equal results, indicating that it tends to overpredict the needed TAs. The other two heuristics, $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$, primarily select TA sets that are a subset of the minimal set and, thus, tend to underpredict the needed TAs.

	$H(\mathcal{A})$	$H(\mathcal{A}_l)$	$H(\mathcal{A}_c)$	$H(\mathcal{A}_l) + H(\mathcal{A}_c)$
Superset	151	25	24	48
Equal	67	107	90	156
Subset	28	117	114	31
Other	3	0	21	14
Total	249	249	249	249

Table 3. TA set comparisons

In evaluating performance, it is misleading to compare spurious inconclusive cases with conclusive cases, since the inconclusive cases terminate the state-space search prematurely. Since $H(\mathcal{A}_l)$ and $H(\mathcal{A}_c)$ tended to be underpredicting heuristics that returned spurious inconclusive results almost 50% of the time, we do not consider them in the performance comparison. Thus, for performance, we restrict the evaluation to $H(\mathcal{A})$ and $H(\mathcal{A}_l) + H(\mathcal{A}_c)$, the two heuristics that had the highest conclusive percentages. Moreover, we also exclude any subject where one of these two heuristics selects a TA set that leads to inconclusive results. Basically, we do not want to compare performance between two analyses where one of them terminates without examining the whole state space while the other does not. As a result, there are 204 subjects in the performance comparison.

Thus in considering performance, we compare $H(\mathcal{A})$, which returned the highest percentage of conclusive results, with $H(\mathcal{A}_l) + H(\mathcal{A}_c)$, which returned the highest percentage of TA sets that matched the minimal set. If the additional TAs selected by $H(\mathcal{A})$ do not significantly degrade performance, then this heuristic would be preferable. If they do significantly degrade performance then the analyst might prefer to use $H(\mathcal{A}_l)+H(\mathcal{A}_c)$ and resort to other approaches for improving precision if necessary.

Figure 3 and Figure 4 show the runtime ratios and the space ratios, respectively, for the 204 subjects for both heuristics. In these figures the subjects have been ordered according to their performance on heuristic $H(\mathcal{A})$, since this heuristic had the greatest performance variability. Note that the vertical scale is logarithmic in each of these figures.

Figure 3 shows that nearly all the runtime ratios of $H(\mathcal{A})$ are larger than or equal to the corresponding runtime ratios of $H(\mathcal{A}_l) + H(\mathcal{A}_c)$. The worst runtime ratio of $H(\mathcal{A})$ is larger than 2,000 and thus is not completely shown in Figure 3. Such a large runtime increase would be problematic, especially for problems that are larger than our examples. Heuristic $H(\mathcal{A}_l)+H(\mathcal{A}_c)$ had runtime ratios around 1, with the largest ratio being less than 2. Thus, for this heuristic, the cost of automatic TA selection might be worth the expected additional runtime cost.

In Figure 4, we have a similar observation. The space ratios for $H(\mathcal{A})$ are always equal to or larger than those for $H(\mathcal{A}_l) + H(\mathcal{A}_c)$. For $H(\mathcal{A})$, the worst space ratio, which again is not completely shown in the figure, is more than 26,000. (Recall that there are also four subjects that get out-of-memory results with this heuristic.) Almost all space ratios of $H(\mathcal{A}_l) + H(\mathcal{A}_c)$, on the other hand, are close to 1.

Thus, although $H(\mathcal{A})$ has the highest conclusive percentage, the performance cost is probably too high for this heuristic to be routinely applied. The heuristic $H(\mathcal{A}_l) + H(\mathcal{A}_c)$, however, produced surprisingly good results in terms of precision, with a relatively small decrease in performance.

The experiment described above assumed that the analyst had first selected a set of VAs and CAs. This seems like a reasonable assumption since, in our experience, analysts often can anticipate which VAs and CAs to include. We wondered how this assumption affected our result, however. In particular, if we did not include information about the VAs and CAs, would the $H(\mathcal{A}_l) + H(\mathcal{A}_c)$ heuristic return drastically different TA sets? Thus, we did not include any VA or CA constraints and recomputed the TAs that would be selected by all four of the heuristics. Table 4 shows the TA set comparisons that result when VAs and CAs are not included. Comparing Table 4 with Table 3 shows that all the heuristics are affected by this change except $H(\mathcal{A}_c)$; this is because the events in VAs and CAs are usually attached to local nodes, which are not considered by this heuristic. The affected heuristics select fewer TAs, since the event alphabet is smaller, and thus the number of superset and equal sets decreased whereas are the number of subset and other sets increased. Since most of the subjects do require VA

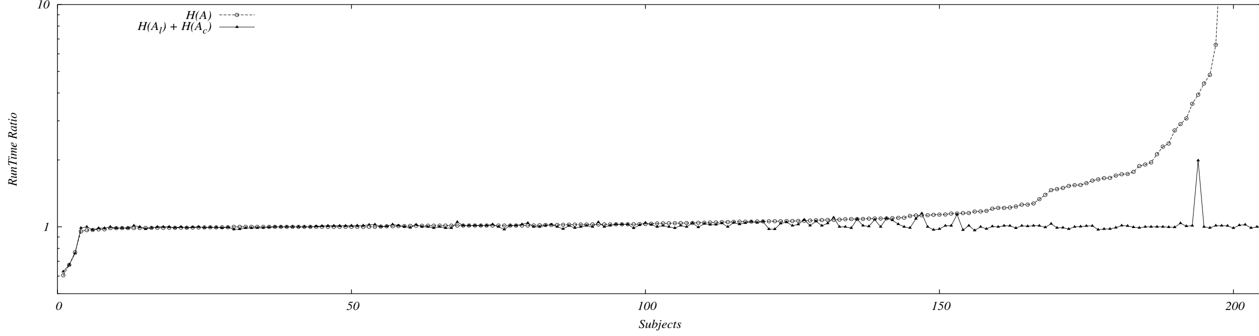

Figure 3. Runtime ratios

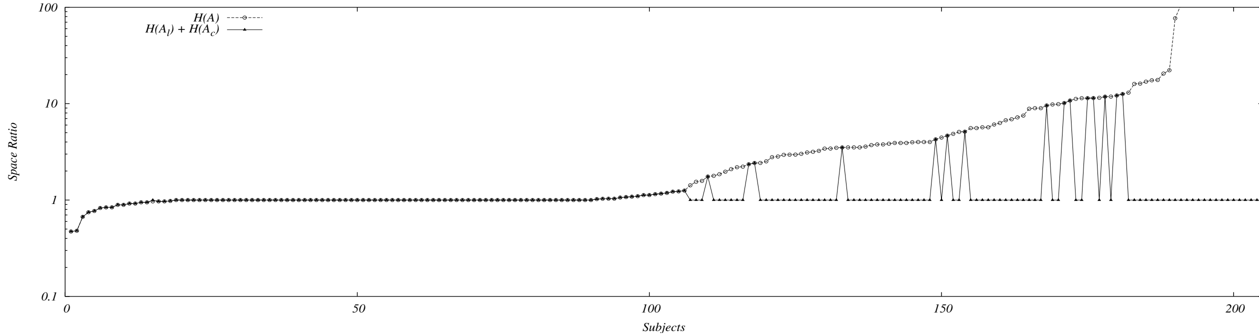

Figure 4. Space ratios

or CA constraints for conclusive analysis, we do not compare performance here. Note however that $H(\mathcal{A}_l) + H(\mathcal{A}_c)$ still returns 122 equal sets for the 249 subjects. Thus, even when VAs and CAs are not considered, this heuristic would be useful in selecting TAs to be included in the verification.

	$H(\mathcal{A})$	$H(\mathcal{A}_l)$	$H(\mathcal{A}_c)$	$H(\mathcal{A}_l) + H(\mathcal{A}_c)$
Superset	138	21	24	45
Equal	53	60	90	122
Subset	46	168	114	61
Other	12	0	21	21
Total	249	249	249	249

Table 4. Set comparisons without VAs or CAs

6 Related Work

Our approach is concerned with finding good heuristics to help refine the system model used during analysis. Thus, we consider related work in finite-state verification on model refinement and on the use of heuristics.

Model abstraction is a key technique that can be used to combat the state-explosion problem. Data abstraction and predicate abstraction, for example [2, 4, 8, 11, 13, 15], try to collapse the states in a model by finding equivalent classes of data values based on language homomorphisms or predicates. Slicing-based abstraction, such as [11, 15], applies dependence analysis to eliminate data and control flow information that is irrelevant to the property being checked. All these approaches are designed to remove information from the model. In contrast, our approach starts with a small, but coarse, model, and then adds information to refine the model.

Counterexample-guided abstraction refinement approaches, such as [2, 3, 16], are similar to our approach in that information is added to an imprecise model to improve precision. With these approaches, the information to be added is extracted from a spurious counterexample path. Our work also eliminates infeasible paths from the model, but our approach can be applied before any verification has been attempted and thus before a counterexample path has been found. Our work can be thought of as an eager refinement approach, whereas counterexample-guided refinement is a demand-driven approach.

Heuristics have been used in finite-state verification to help find counterexample paths. The goals have been to find shorter paths or to improve performance, or both [5, 12, 14, 28]. The heuristics have been based on the property or on the model, as with our heuristics, as well as on violation-state information. Instead of guiding the counterexample

search, the heuristics we have developed are used to help refine the model, which should eliminate some infeasible paths but may not improve performance or lead to shorter counterexample paths.

7 Conclusion

One special feature of FLAVERS is that it allows an analyst to refine the model incrementally by adding constraints. The choice of constraints involves a delicate trade-off between precision and cost: if the right constraints are not selected, the analysis will be inconclusive, but selecting too many constraints tends to increase the cost of analysis significantly. While some useful constraints are relatively easy for the analyst to identify, the TA constraints that eliminate spurious interleavings of events from different tasks are not. Selection of a suitable set of TAs has thus typically been an iterative process, in which the analyst repeatedly runs the analysis, finds spurious counterexample paths, and adds TAs to eliminate those paths. Automated support for selecting a good set of TAs, one that provides the necessary precision without incurring too high a cost, would be of substantial value

In this paper, we have presented four heuristics for TA prediction in FLAVERS for Ada programs. All these heuristics are based on the subject alphabet. One of these heuristics, $H(\mathcal{A}_l) + H(\mathcal{A}_c)$, seems to provide a very good balance of precision and cost. The TAs selected with this heuristic are sufficient for conclusive analysis in more than 80% of the cases we studied, and the added cost in time and space, when compared to a manually identified minimal set of TAs, is usually quite small. Moreover, it gives fairly good results even without information about the VAs and CAs that the analyst would select. This heuristic seems therefore to offer an excellent starting point for TA selection, in most cases eliminating the need for manual selection altogether.

There are a number of additional directions that we intend to explore in the future. There were still a little over 18% of the cases where the TAs selected by $H(\mathcal{A}_l)+H(\mathcal{A}_c)$ were not sufficient to obtain a conclusive result. We are interested in ways to refine the heuristic or to add additional heuristics that might improve this. We will also investigate additional and larger examples to see if the performance of the heuristic holds up when it is applied to other classes of systems and properties. Analysis of large systems is a major undertaking, and we would have no way of knowing whether we had ever obtained a truly representative sample of systems and properties, so of course we do not expect to obtain completely definitive results. We would be especially interested in identifying classes of systems for which particular approaches to TA selection are well suited.

We are also interested in the problem of automatically selecting VAs. An analyst with a good understanding of a system and the property to be verified can often select appropriate VAs fairly easily. With very large systems, however, or ones for which the analyst does not fully understand the details of the interaction between components, heuristics for VA selection would also be of significant value.

As discussed earlier, all of the properties checked in this study held for their respective systems. The comparison of performance is significantly trickier in the case where the properties do not hold, and previous research [5] has shown that different algorithms may be appropriate if FLAVERS is applied early in development when many bugs remain and the properties being checked are not likely to hold in most cases. We intend to investigate the performance of these heuristics in such cases and to look for other heuristics that may give better performance. We are also particularly interested in applying heuristics to the algorithms used by FLAVERS to search for counterexamples, as for instance in [12], in the hope of obtaining more useful counterexamples or finding a counterexample more quickly.

Finally, FLAVERS/Ada is an application of the FLAVERS approach to Ada programs. We have previously described how the approach can be extended to Java programs [21, 22] and are currently building tools, based on the Bandera toolset [7], for extracting FLAVERS models from Java source code. As soon as those tools are complete, we will investigate the applicability of similar heuristics for constraint selection for analyzing Java programs.

Acknowledgments

We are very grateful to Heather Conboy and Jamieson Cobleigh for their assistance in collecting the experimental subjects and answering questions about the FLAVERS implementation, and to Shangzhu Wang and Nathan Jokel for reviewing earlier drafts of this paper.

This research was partially supported by the U.S. Army Research Laboratory and the U.S. Army Research Office under Agreement DAAD190110564, by the U.S. Department of Defense/Army Research Office under Grant No. DAAD19-03-1-0133, and by the National Science Foundation under Grant No. CCR-0205575.

References

[1] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Păsăreanu, and S. F. Siegel. Comparing finite-state verification techniques for concurrent software. TR UM-CS-1999-069, Department of Computer Science, U. of Massachusetts Amherst, Nov. 1999.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. of the 8th Int. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 103–122. Springer-Verlag, May 2001.

[3] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, Sep. 2003.

[4] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM Trans. on Programming Languages and Systems*, 16(5):1512–1542, Sep. 1994.

[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 *Proc. of the 23rd Int. Conf. on Software Engineering*, pages 37–46. IEEE Computer Society, 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 on Software Testing and Verification*, 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 *Proc. of the 22nd Int. Conf. on Software Engineering*, pages 439–448, June 2000.

[8] S. Das, D. L. Dill, and S. Park. Experience with predicate abstraction. In *11th Int. Conf. on Computer-Aided Verification*, volume 1633 of *LNCS*, pages 160–171. Springer-Verlag, July 1999.

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

[10] M. B. Dwyer, L. A. Clarke, J. M. Cobleigh, and G. Naumovich. Flow analysis for verifying properties of concurrent systems. TR UM-CS-2003-030, Department of Computer Science, U. of Massachusetts Amherst, Sep. 2003.

[11] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, H. Zheng, and W. Visser. Tool-supported program abstraction for finite-state verification. In *Proc. of the 23rd Int. Conf. on Software engineering*, pages 177–187. IEEE Computer Society, May 2001.

[12] S. Edelkamp, A. L. Lafuente, and S. Leue. Directed explicit model checking with HSF-SPIN. In *Proc. of the 8th Int. SPIN Workshop on Model Checking of Software*, volume 2057 of *LNCS*, pages 57–79. Springer-Verlag, May 2001.

[13] S. Graf and H. Saidi. Construction of abstract state graphs with PVS. In *9th Int. Conf. on Computer Aided Verification*, volume 1254 of *LNCS*, pages 72–83. Springer-Verlag, June 1997.

[14] A. Groce and W. Visser. Model checking Java programs using structural heuristics. In *Proc. of the Int. Symp. on Software Testing and Analysis*, pages 12–21. ACM Press, July 2002.

[15] C. Heitmeyer, J. Kirby, Jr., B. Labaw, M. Archer, and R. Bharadwaj. Using abstraction and model checking to detect safety violations in requirements specifications. *IEEE Trans. on Software Engineering*, 24(11):927–948, Nov. 1998.

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

[17] G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison Wesley Professional, MA, 2003.

[18] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: a unified model. *Acta Informatica*, 28(2):121–163, Dec. 1990.

[19] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, MA, 1993.

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

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

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

[23] G. Naumovich, L. A. Clarke, and J. M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proc. of the ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 57–65, Sep. 1999.

[24] K. M. Olender and L. J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis generation. *IEEE Trans. on Software Engineering*, 16(3):268–280, Mar. 1990.

[25] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Proc. of the 7th European Software Engineering Conf. held jointly with the 7th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 235–252. Springer-Verlag, Sep. 1999.

[26] J. Tan, G. S. Avrunin, and L. A. Clarke. Heuristic-based model refinement for FLAVERS. TR UM-CS-2003-029, Department of Computer Science, U. of Massachusetts Amherst, Sep. 2003.

[27] O. Tkachuk and M. B. Dwyer. Adapting side effects analysis for modular program model checking. In *Proc. of the 9th European Software Engineering Conf. held jointly with the 10th ACM SIGSOFT Int. Symp. on Foundations of Software Engineering*, pages 188–197. ACM Press, Sep. 2003.

[28] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of the 35th Design Automation Conf.*, pages 599–604, June 1998.