

# The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification

Jamieson M. Cobleigh, Lori A. Clarke, Leon J. Osterweil  
Laboratory for Advanced Software Engineering Research  
Department of Computer Science  
University of Massachusetts Amherst  
Amherst, MA 01003-6410  
+1 413 545 2013  
{jcobleig, clarke, ljo}@cs.umass.edu

## Abstract

*Finite state verification is emerging as an important technology for proving properties about software. In our experience, we have found that analysts have different expectations at different times. When an analyst is in an exploratory mode, initially formulating and verifying properties, analyses usually find inconsistencies because of flaws in the properties or in the software artifacts being analyzed. Once an inconsistency is found, the analyst begins to operate in a fault finding mode, during which meaningful counter example traces are needed to help determine the cause of the inconsistency. Eventually systems become relatively stable, but still require re-verification as evolution occurs. During such periods, the analyst is operating in a maintenance mode and would expect re-verification to usually report consistent results. Although it could be that one algorithm suits all three of these modes of use, the hypothesis explored here is that each would be best served by an algorithm optimized for the expectations of the analyst.*

## 1. Introduction

Finite state verification (FSV) is emerging as an important technology for proving properties about software systems. Although there are many approaches for doing FSV, most support a specification formalism for representing properties, a model for representing the software system, a reasoning engine that attempts to verify if a property holds on all possible executions, or traces, through the model, and a counter example generator that provides traces through the model if the model is not found to be consistent with the property.

In our experience with FSV, we have found that analysts

have different expectations at different times. At first analysts are in an *exploratory* mode, trying to formulate properties and attempting to prove them. Whether analyzing a design or code artifact, the analyst must decide on the important properties to be evaluated and then represent those properties in the specification notation. Initially the designs or code artifacts are likely not to be consistent with a formulated property because of defects in the property, defects in the artifact, or both. Only after several attempts to formulate and verify a property, and to correct problems that arise, is an analyst likely to obtain a result indicating that the property is consistent with the system. Thus, an analyst in an exploratory mode would expect analysis to usually report an inconsistency between the property and the model. Therefore during this mode, it would make sense to use a reasoning engine optimized to find inconsistent results.

After learning that analysis results are inconsistent, an analyst usually immediately moves to a debugging or *fault finding* mode in which the analyst is seeking the cause of the problem. Finite state verification approaches usually provide a trace or path through the model (or the corresponding path through the original software system), but these paths are sometimes long and convoluted [2, 9]. Complicated paths make it more difficult to track down the actual cause of the inconsistency. Thus, when an analyst is in fault finding mode, it makes sense to use a reasoning engine optimized to produce short paths or user-guided paths that reveal the inconsistency.

A more mature software system ideally has many verified properties. If the system is modified, the analyst should re-verify that the remaining relevant properties still hold. If the system has not been modified too extensively, the analyst should expect that most of the previously verified properties would continue to be consistent with the evolved software system. Thus, when an analyst is in this *maintenance*

mode, it makes sense to use a reasoning engine optimized to produce consistent results.

Although one algorithm might suit all three types of activities, our hypothesis is that each would be best served by an algorithm optimized for the expectations of the analyst. We have thus been experimenting with different algorithms to determine how well they meet these expectations. Although the algorithms we explored are specific for the FLAVERS prototype [6], the general underlying concepts can be applied to many of the other finite state verification approaches. Thus, we believe that our results might generalize to other finite state verification approaches as well. In this paper we describe the algorithms we considered and the results of our experimentation.

## 2. FLAVERS

FLAVERS (**FL**ow Analysis for **VER**ification of **Sys**tems) is a static analysis approach that can verify user specified properties of sequential and concurrent systems. Like all automated verification systems, FLAVERS requires an abstract model of the computation upon which to base the analysis. The model FLAVERS uses is based on annotated *Control Flow Graphs* (CFG). Annotations are placed on nodes of the CFGs to represent events that occur during execution of the actions associated with a node. For concurrent systems, FLAVERS uses a *Trace Flow Graph* (TFG), which consists of a collection of CFGs with some additional intertask edges and nodes to represent the concurrency. Our experiments used FLAVERS to analyze concurrent Ada programs, so some intertask edges represent intertask communication via rendezvous. The other intertask edges are *May Immediately Precede* (MIP) edges that represent potential interleavings of events in different tasks [17]. A CFG, and thus a TFG, over-approximates the sequences of events that can occur when executing a system.

FLAVERS requires that a property to be checked be represented as a *Finite State Automaton* (FSA). FLAVERS uses an efficient state propagation algorithm to determine whether all potential behaviors of the system being analyzed are consistent with the property. FLAVERS will either return *conclusive*, meaning the property being checked holds for all possible paths through the TFG, or *inconclusive*, meaning FLAVERS found some path through the TFG that causes the property to be violated. FLAVERS analyses are conservative, meaning FLAVERS will return conclusive results only when the property holds for all TFG paths. FLAVERS returns inconclusive results either because there is an execution that actually violates the property or because the property is only violated on paths through the TFG that do not correspond to actual system executions. These so called *infeasible paths* result from the imprecision of the model, and their effects can be eliminated by introducing

*feasibility constraints*, also represented as FSAs. For example, an analyst might introduce an FSA to keep track of the value of a program variable. An analyst might need to iteratively add feasibility constraints and observe the analysis results several times before determining whether a property is conclusive. Feasibility constraints give analysts some control over the analysis process by letting them determine what parts of a system need to be modeled more precisely.

FLAVERS' state propagation has worst-case complexity that is  $\mathcal{O}(N^2 \cdot |S|)$ , where  $N$  is the number of nodes in the TFG, and  $|S|$  is the product of the number of states in the property and the number of states in each of the feasibility constraints. In our experience, a large class of important properties can be proved by using only a small set of constraints. Experimental results seem to indicate that the cost of solving most problems is low order polynomial, often sub-cubic, in the size of the system. Thus, FLAVERS has the potential to scale to handle realistic software systems.

## 3. Basic definitions

Formally, a TFG is a labeled directed graph  $G = (N, E, n_{initial}, n_{final}, \Sigma_G, L)$  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 respectively,  $\Sigma_G$  is an alphabet of event labels associated with the TFG, and  $L : N \rightarrow \Sigma_G$  is a function mapping nodes to their labels.

Formally, an FSA is a five-tuple,  $F = (S, \delta, s^0, A, \Sigma)$  where  $S$  is a finite set of states,  $\Sigma$  is a finite alphabet,  $\delta : S \times \Sigma \rightarrow S$  is a total transition function,  $s^0 \in S$  is a unique start state, and  $A \subseteq S$  is a set of accepting states. Every property and feasibility constraint is specified as an FSA. Each constraint, however, has an extra state known as the *constraint violation state*,  $v$ , that represents an unrecoverable constraint violation and is a non-accepting state with only self-loop transitions.

FLAVERS uses a fixed point algorithm that propagates tuples, each representing a state in the property and each of the constraints, through the model of the system [20]. Suppose we wish to verify a property  $P = (S_P, \delta_P, s_P^0, A_P, \Sigma_P)$  over a TFG  $G = (N, E, n_{initial}, n_{final}, \Sigma_G, L)$  using a set of constraints  $C_1, \dots, C_k$  where  $C_i = (S_{C_i}, \delta_{C_i}, s_{C_i}^0, A_{C_i}, \Sigma_{C_i}, v_{C_i})$ . The set of all tuples is  $\mathcal{T} = S_P \times S_{C_1} \times \dots \times S_{C_k}$ . A tuple is any  $t \in \mathcal{T}$ . Define the *initial tuple* as the tuple  $T^0 = (s_P^0, s_{C_1}^0, \dots, s_{C_k}^0)$ . Define a transition function  $\Delta : \mathcal{T} \times N \rightarrow \mathcal{T}$  as follows

$$\Delta((s_P, s_{C_1}, \dots, s_{C_k}), n) = (s'_P, s'_{C_1}, \dots, s'_{C_k})$$

where

$$s'_P = \delta_P(s_P, L(n)) \text{ and } \forall 1 \leq i \leq k : s'_i = \delta_{C_i}(s_{C_i}, L(n))$$

This transition function takes a tuple and a TFG node and produces a new tuple by determining the effect the label on the node has on each FSA in the tuple. In verifying a property, we associate a set of tuples with each node. The initial node starts with  $T^0$  associated with it. From here, tuples are propagated forward through the TFG using the transition function  $\Delta$  to compute the tuples associated with the nodes of the TFG. To verify a property, we need to consider every path in the TFG, making state propagation a forward-flow, any-path data flow problem [15].

State propagation eventually reaches a fixed point where no new tuples can be associated with any nodes. At this point, the results of the verification can be determined. FLAVERS is concerned with terminating program executions, so only the tuples on  $n_{final}$  are examined. The tuples on the final node are all of the combinations of the states of the property and the states of the constraints that occur on terminating program executions. We look for violating tuples on the final node. A *violating tuple* is one for which the property automaton is in a non-accepting state, representing a property violation, and every constraint is in an accepting state, ensuring that all feasibility constraints are satisfied. More formally, a violating tuple is  $t = (s_P, s_{C_1}, \dots, s_{C_k})$  where  $\forall 1 \leq i \leq k : s_{C_i} \in A_{C_i}$  and  $s_P \notin A_P$ . If there are violating tuples on the final node, then the property does not hold and the result is inconclusive. Otherwise, there are no ways that the property can be violated, so the property holds and the result is conclusive.

As noted above, each feasibility constraint has a state called the constraint violation state,  $v$ . When propagating tuples, if a tuple  $t$  returned by  $\Delta$  has any constraint  $C_i$  in its constraint violation state, then  $t$  need not be propagated forward. The state  $v$  has only self-loop transitions, so any tuple  $t'$  that reaches the final node as a result of repeated applications of  $\Delta$  to  $t$  will have  $C_i$  in its constraint violation state. Thus, when we examine the tuples on  $n_{final}$ , we will discard  $t'$  since it corresponds to a infeasible path. Consequently, a tuple with a constraint in its constraint violation state is discarded as soon as it is created. Let  $\mathcal{T}_V$  be the set of all such tuples with constraint violations.

$$\mathcal{T}_V = \{(s_P, s_{C_1}, \dots, s_{C_k}) \mid \exists 1 \leq i \leq k : s_{C_i} = v_{C_i}\}$$

With these formal definitions, we can provide a state propagation algorithm to verify a property  $P$  over a TFG  $G$  with constraints  $C_1, \dots, C_k$ . Meta-Algorithm MA, shown in Figure 1, is the state propagation meta-algorithm. It uses a worklist  $Wlist$  to keep track of the nodes to be processed. With each node  $n$  in the TFG it associates a set of tuples, held in  $Tuples[n]$ . The initial tuple is associated with the initial node, which is placed on the worklist. The algorithm iterates until the worklist is empty. During each iteration, a node  $n$  is removed from the worklist (line 2). For each successor  $m$  of  $n$ , first the original set of tuples on  $m$  is

Initially:

$$Wlist := n_{initial}$$

$$Tuples[n] := \begin{cases} \emptyset & \text{if } n \neq n_{initial} \\ \{T^0\} & \text{if } n = n_{initial} \end{cases}$$

Main Loop:

- (1) while  $Wlist \neq \emptyset$  do
- (2)    $n$  is a node removed from  $Wlist$
- (3)   foreach  $m$  a successor of  $n$  do
- (4)      $temp := Tuples[m]$
- (5)      $Tuples[m] := \left( Tuples[m] \bigcup_{t \in Tuples[n]} \Delta(t, m) \right) \setminus \mathcal{T}_V$
- (6)     if  $Tuples[m] \neq temp$  then
- (7)       insert  $m$  into  $Wlist$
- end if
- done
- done

Figure 1. Meta-Algorithm MA

saved (line 4), so the algorithm can tell if new tuples are later added to  $m$ . Then every tuple on  $n$  is propagated via the transition function  $\Delta$  to  $m$ , removing any tuples that have a constraint in a constraint violation state (line 5)<sup>1</sup>. Finally, the set of tuples on  $m$  is compared to the saved set (line 6). If they are not the same, then at least one tuple was added to  $m$ , and  $m$  is put on the worklist (line 7). After processing all successors of  $n$ , control returns to the outer loop to see if the worklist is empty (line 1). When MA terminates,  $Tuples[n_{final}]$ , the set of tuples associated with the final node, is examined. If there are violating tuples, the property does not hold, otherwise it does.

To create a counter example trace, we need to find a path through the TFG that starts at the initial node, ends at the final node, and results in a property violation. More formally, we want a finite path  $n_1, n_2, \dots, n_l$ , such that  $n_1 = n_{initial}$ ,  $n_l = n_{final}$  and there exist tuples  $t_1, t_2, \dots, t_l$  such that  $t_1 = T^0$ ,  $t_l$  is a violating tuple, and  $\forall 1 < i \leq l : t_i = \Delta(t_{i-1}, n_i)$ .

## 4. Algorithmic ontology

MA is a high level description of a general class of state propagation algorithms. Many details, such as the order in

<sup>1</sup>Clever bookkeeping can improve the efficiency of line 5 of MA. As presented, each tuple of node  $n$  is propagated to node  $m$  on every loop iteration. In our implementation, each node keeps track of what tuples it has propagated to its successors so when line 5 is reached only new tuples are propagated.

Initially:

$$Wlist = \{m \mid (n_{initial}, m) \in E\}$$

Main Loop:

- (3)  $temp := Tuples[n]$
- (4) foreach  $p$  a predecessor of  $n$  do
- (5)  $Tuples[n] := \left( Tuples[n] \bigcup_{t \in Tuples[p]} \Delta(t, n) \right) \setminus \mathcal{T}_V$
- done
- (6) if  $Tuples[n] \neq temp$  then
- (7)   foreach  $m$  a successor of  $n$  do
- (8)     insert  $m$  into  $Wlist$
- done
- end if

**Figure 2. Changes to MA for pull**

which nodes are selected for processing (lines 2 and 3), are not fully specified. These details may have important effects upon such issues as efficiency. To determine the effects of these details upon the issues of concern in this paper we performed some experiments. These experiments can be viewed as analyses of the effects of making different combinations of choices along four dimensions of variability in the details of MA.

One dimension entails adding appropriate details to lines 2 and 7 of the algorithm of Figure 1. By varying this dimension,  $Wlist$  could be managed either as a stack or a queue. When managed as a stack the effect is a depth first search of the graph nodes. When managed as a queue, the effect is breadth first. We expected depth first order to improve cache performance; since recently inserted nodes are examined first, and are likely to still be cached. Additionally, a depth first search explores one path deeply, and thus may find a violation quickly if it serendipitously picks nodes that lead to some violation. However, if all violations go through a small set of nodes that are not encountered on the early selected paths or these nodes get stuck on the bottom of the worklist, then it may be worse than breadth first search. Moreover, breadth first search will find a shortest path, whereas depth first makes no guarantees about the length of the counter example it will find.

A second dimension entails elaborating on line 3. This can alter the order successors are visited. This order may be varied, perhaps even for each time a node is removed from the worklist. Since this order affects the search order, it can alter the performance of our algorithms.

A third dimension entails choosing between a push and a pull algorithm. In MA, when a node is removed from the worklist on line 2, it has at least one new tuple to be prop-

agated to its children. This algorithm *pushes* tuples from a parent to its children. An alternative would be to insert a node onto the worklist when one of its parents has at least one new tuple to be propagated to it. This can be done by replacing lines 3-7 in Figure 1 with lines 3-8 in Figure 2. Here, the tuples are *pulled* by a child from its parents and that child then inserts its successors onto the worklist. These two approaches would seem to yield differing performance characteristics.

A fourth dimension entails using a different graph model. MA inserts nodes on a worklist and considers a node to have a set of tuples associated with it. Thus, this algorithm works over the graph  $(N, E)$ , defined over the *node space*. Alternatively, we might base our work on a graph whose nodes are drawn from  $N \times \mathcal{T}$ , or *node-tuple space*. In this space, node-tuples are placed on the worklist instead of nodes. In this case, verification entails finding a path through  $N \times \mathcal{T}$  that starts at  $\langle n_{initial}, T^0 \rangle$  and ends at  $\langle n_{final}, t_{viol} \rangle$ , where  $t_{viol}$  is any violating tuple. If no such path exists, the property holds. Walking through node space is a coarsening of the view of the problem as a walk through node-tuple space. As a result, there is the typical space versus computation tradeoff. The number of nodes in node space is smaller than in node-tuple space but more processing is needed for each node removed from the worklist. Also, information about which tuple propagated information into which other tuples is lost. As a result, counter examples cannot easily be constructed in node space. It seemed clear that choosing between a node space algorithm and a node-tuple space algorithm would produce important differences in performance characteristics.

## 5. The algorithms

Three of the dimensions in our ontology are binary, yielding 8 possible algorithms. The remaining dimension, the order of considering successors, can have many alternatives. This yields over 16 possible algorithms, some of which do not make sense, however, while others have little affect on behavior.

First, none of the pull algorithms places node-tuples on the worklist. In node space, we have done a coarsening, so it is possible to add a node to the worklist and have it derive its new tuples from its parents. In node-tuple space, to add a node-tuple to the worklist, the associated tuple must first be computed, hence a pull version of the algorithm does not make sense.

Second, the order in which items are added to the worklist does not greatly affect the performance of the breadth first algorithms. Suppose two items  $s_1$  and  $s_2$  are added to the worklist in that order. With a depth first worklist,  $s_2$  will be removed, and its successors will be added to the worklist. As a result,  $s_1$  remains trapped on the worklist until every

Algorithm	Finds Paths	Push or Pull	Search Order	Worklist Contents
Push DFS'	No	Push	DFS'	Nodes
Push BFS'	No	Push	BFS'	Nodes
Pull DFS'	No	Pull	DFS'	Nodes
Pull BFS'	No	Pull	BFS'	Nodes
Push DFS Std	Yes	Push	DFS	Node-Tuples
Push DFS Rev	Yes	Push	DFS	Node-Tuples
Push DFS Wrp	Yes	Push	DFS	Node-Tuples
Push BFS	Yes	Push	BFS	Node-Tuples
A*	Yes	Push	BFS	Node-Tuples

Table 1. Summary of algorithms

path through  $s_2$  is considered. The order of consideration of  $s_1$  and  $s_2$  can make a large difference in when they are considered. With a breadth first order,  $s_1$  will be removed from the worklist, followed immediately by  $s_2$ . The order these two items are added to the worklist has a minimal impact on when they are removed. As a result, the order in which successor nodes are added to the worklist is only considered for the depth first worklist algorithms.

Finally, in node-space, the order in which successor nodes are added to the worklist does not greatly affect the performance of the depth first algorithms. Since a node is not inserted onto the worklist if it is already on the worklist, the worklist is bounded in size. Thus, items do not get trapped on the worklist as easily. Additionally, the effect of all new tuples is considered when a node is removed from the worklist. For example, with push algorithms, several tuples may be propagated to successors when a node is removed from the worklist. This helps drive the algorithm forward and helps prevent tuples from being ignored for a long time.

Table 1 shows the algorithms that we considered.

The first two of these algorithms, Push DFS' and Push BFS' are almost exactly the algorithm shown in Figure 1. They insert nodes onto the worklist and push tuples from a parent to its children. As noted above, nodes already on the worklist are not added to it again. This optimization puts an upper bound on the worklist size, but changes the behavior from strict depth first or breadth first. We use DFS' and BFS' to represent this small modification.

The next two algorithms, Pull DFS' and Pull BFS' are analogous to Push DFS' and Push BFS' except a node pulls its tuples down from its parents. These first four algorithms all walk through node space. As previously discussed, this coarsening makes constructing a counter example more difficult, so none of these four algorithms produces counter examples. The remaining algorithms all search through node-tuple space and find counter examples if they exist.

The next three algorithms in Table 1 all use a depth first worklist and insert node-tuples onto it. With a worklist

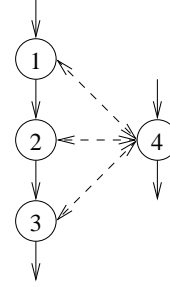


Figure 3. Task automata example

based on node-tuples, a true depth first worklist is used, since a node-tuple is never revisited once it is processed.

Push DFS Std, Push DFS Rev, and Push DFS Wrp vary only in the order in which they consider successors. Push DFS Std uses a *standard* ordering, where nodes are considered in the order the successor nodes occur in the TFG (intratask edges before intertask edges). Push DFS Rev uses a *reversed* ordering that simply reverses the order of standard. Push DFS Wrp uses a *wrapped* ordering, that attempts to add variety to the search. Suppose a node,  $n$  has  $m$  successors. The successors are returned in the same order as standard, except the  $k^{th}$  time we encounter a node, we return the  $k \bmod m$  successor first, then the  $(k + 1) \bmod m$  successor, and so on.

The next algorithm, Push BFS is similar to Push DFS Std except Push BFS uses a breadth first worklist. We found that breadth first search often required more time to find a path than depth first search, but returned shortest paths, which can be important in fault finding mode.

Breadth first search uses no information about the problem when searching. An *informed search*, such as A\*, which makes use of problem specific information, might do better. A\* search uses a heuristic that estimates the cost from a node in the search space to a goal node. Items on the worklist are ordered by the sum of this heuristic estimate and the cost to reach them from the initial node. This sum is an estimate of the total cost of the shortest path that goes through a node. If the heuristic is *admissible*, meaning it never overestimates the cost to a goal node, then A\* is guaranteed to find a shortest path [5]. Since the worklist is ordered by the estimate of the cost, the worklist must be a priority queue.

To make use of A\* search, a heuristic is needed to estimate cost. Our heuristic is based upon *task automata* feasibility constraints, which are generated automatically by FLAVERS and are used to represent the legal flow through a single task. Consider the TFG fragment in Figure 3. Suppose nodes 1, 2, and 3 are adjacent nodes in one task, and node 4 is in another task. The solid arrows represent intratask flow, while the dashed edges are MIP edges, repre-

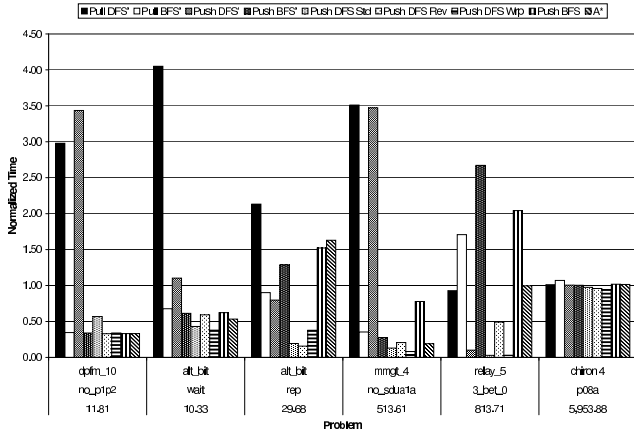


Figure 4. Inconclusive problems

senting possible intertask flows. In this fragment, one possible path is  $1 \rightarrow 4 \rightarrow 3$ , which is not feasible since it skips node 2. A task automaton can prevent this by ensuring that nodes within a given task are visited in a legal order. It does this by keeping track of the last node visited in a task, so a task automaton has one state for every node in the task. As a result, we can use a task automaton to determine the minimum number of nodes within a task that need to be traversed to reach the final node of the TFG. Summing this estimate for each task automaton in a tuple, yields an estimate of the number of TFG nodes to be traversed before the final node of the TFG can be reached<sup>2</sup>. Since this heuristic never overestimates the cost to reach a goal state, it is admissible and an A\* search using this heuristic will find a shortest path. We call this algorithm A\*.

## 6. Experimental results

To evaluate our algorithms, we used a suite of Ada programs, mostly drawn from the literature, that we had previously collected for testing FLAVERS. These included the Chiron user interface development system [8], several variants on the dining philosophers problem, a simulation of a gas station [10], a memory management system [7], and some communications protocols [19]. Many of these problems are arbitrarily scalable by adding instances of existing tasks. Additionally, many of the properties required constraints to be conclusively verified, so we derived many of the inconclusive problems by removing constraints from conclusive problems. This yielded a test suite of 220 problems: 109 conclusive, 111 inconclusive. These Ada programs had between 1 and 25 tasks and ranged in size from

<sup>2</sup>The actual implementation is more complicated than this, because of intertask nodes that are used to explicitly represent an Ada rendezvous. These nodes belong to multiple tasks and they must not be multiply counted in the estimate, resulting in a heuristic that is not admissible.

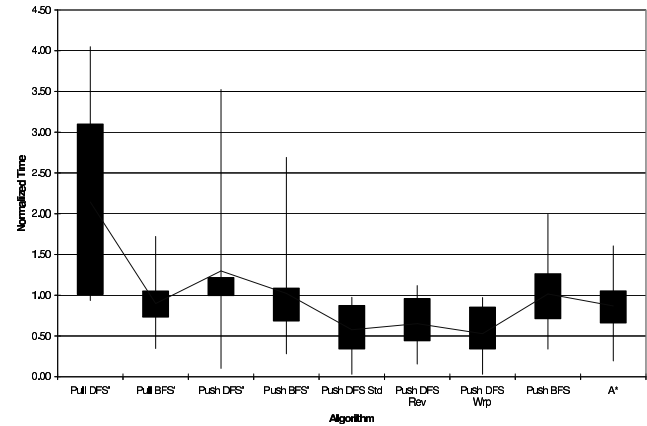


Figure 5. Inconclusive averages

50 to 6,200 lines of code.

All experiments were run on a Sun Enterprise 3500 with two 336 MHz processors and 2 GB of memory, running Solaris 2.6. The FLAVERS toolset is implemented in Java; we ran our experiments using the Sun JDK version 1.1.7.

Space limitations prevent our providing complete data here. Instead, for each algorithm, we show the N problems with longest running times<sup>3</sup>. In addition, since these problems vary widely in their cost, showing the actual running time or path length is not very useful. Instead, we normalized problem times by dividing the actual measured time of all the algorithms by the average time for that problem. Path lengths were normalized by dividing by the length of the shortest path.

### 6.1. Exploratory mode

Recall that in exploratory mode we seek an algorithm that has the best performance over inconclusive problems.

Figure 4 shows the running times for the nine algorithms over six of the largest inconclusive problems. The number below the problem name is the average running time, in seconds, of the nine algorithms on that problem.

In Figure 4, it is easy to see that there is a large variation in the running times among the different algorithms on the same problem. While there appears to be minor variation on the largest of these, the Chiron example, the range of times on this problem was from 88 to 128 minutes. So although the normalization to make the data fit on the plot hides this difference, there is almost a fifty percent change in the running time. These variations are also seen in Figure 5 which shows the averages over the 15 problems that ran longer than 10 seconds for each algorithm. The vertical lines show the range of running times for a given algorithm.

<sup>3</sup>Sometimes this set contained the same property checked on different sized problems. We eliminated the duplicates by removing the smaller sizes since they behaved similar to the larger problems in all cases.

Algorithms		$t$	$P(T \geq t)$	$\sigma$
Push DFS Std	Push DFS Wrp	1.76	0.05	0.03
Push DFS Rev	Push DFS Wrp	2.00	0.03	0.06
Push BFS	A*	2.11	0.03	0.08

**Table 2. t-tests for inconclusive problems**

The black boxes show the range of the middle 50% of the running times, and the horizontal line connects the mean normalized times of the different algorithms.

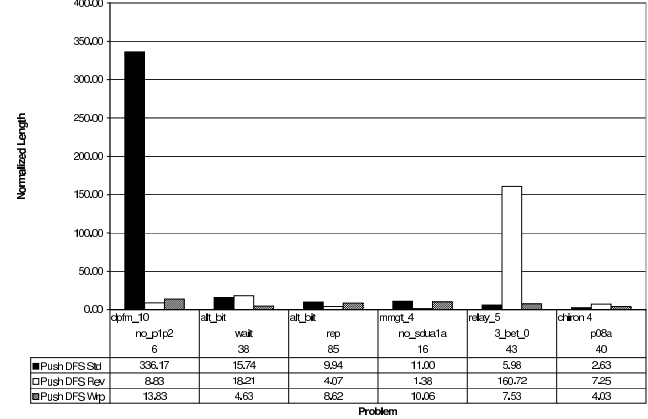
The large variation among the running times of the different algorithms is easy to explain. With a conclusive problem, state propagation explores every derivable node-tuple to ensure that there is no path through the TFG over which the property is violated. With inconclusive problems, once a violating tuple is found, the property is known to be inconclusive and the algorithm can stop. Over the inconclusive problems, chance plays a large factor in determining how large a subset of the reachable node-tuple space an algorithm explores, leading to a wide variation in performance.

From Figures 4 and 5 it can be seen that of the nine algorithms, the ones that deal with node-tuple space perform better than those that deal with node space. Of the five node-tuple space algorithms, the Push DFS algorithms performed better than either Push BFS or A\*. In particular, it appears that Push DFS Wrp has the best performance among the Push DFS algorithms. To confirm this, we performed matched pairs t-tests comparing the two other Push DFS algorithms to Push DFS Wrp. For these t-tests our null hypothesis is that Push DFS Wrp is not faster than the other algorithm and the alternative hypothesis is that it is faster. The results of these tests are shown on the first two rows of Table 2. Since the likelihood of these t-values is small, less than or equal to 5%, we can reject the null hypothesis and say that Push DFS Wrp has the best performance on the inconclusive problems. Thus, we advocate using the Push DFS Wrp algorithm while in exploratory mode since it tends to return inconclusive results the quickest.

## 6.2. Fault finding mode

In fault finding mode, some of the properties are inconclusive and the analyst needs to determine the cause so the fault can be corrected. This task can be aided by the counter examples produced by the verification tool. Having a path is not as important as having a useful path. In our experiment, we use path length as the metric for the usefulness.

Figure 6 shows path length data for the six problems from Figure 4, normalized against the length of the shortest path, which is shown below the problem name in this figure. This figure omits data for the node-space algorithms since they do not return counter examples, and A\* and Push BFS since they both find shortest paths.



**Figure 6. Path lengths**

The data in Figure 6 shows that the Push DFS algorithms have a large variance in the length of their counter examples. We would not expect these algorithms to find short paths since they can make no guarantees about the lengths of the paths they will find. It appears that Push DFS Wrp algorithm, however, tends to find shorter paths than the other two. This is probably because it considers the children of a node in a different order each time it visits that node. As a result, it is not going to loop over the same set of nodes repeatedly, creating a long path that does not make much progress towards the goal. Because of the large variance in path lengths and the small number of problems in our test set, however, we could not come to any statistically significant conclusions to show that Push DFS Wrp was the best Push DFS algorithm in terms of path length.

Thus, one might assume that the Push DFS Wrp algorithm is the best algorithm to use for finding inconclusive results and creating counter examples. If there can only be one algorithm associated with FLAVERS, this indeed might be the case. On average, however, Push DFS Wrp found paths that were 5.88 times longer than the shortest path. Since analyst time is probably of more concern than computing time, having a short path is probably worth the additional computation time of first finding an inconsistency, using the Push DFS Wrp algorithm, and then finding a shortest path, using a shortest path algorithm. Of the two shortest path algorithms we considered, Push BFS and A\*, there is statistical evidence that A\* is faster as shown in Table 2. Even though A\* runs on average 3.87 times slower than Push DFS Wrp, we feel that getting significantly shorter paths is worth this extra time investment.

## 6.3. Maintenance mode

Eventually, the analyst should be in a maintenance mode re-proving properties for systems that have evolved. In this mode, we are interested in algorithms that have the best per-

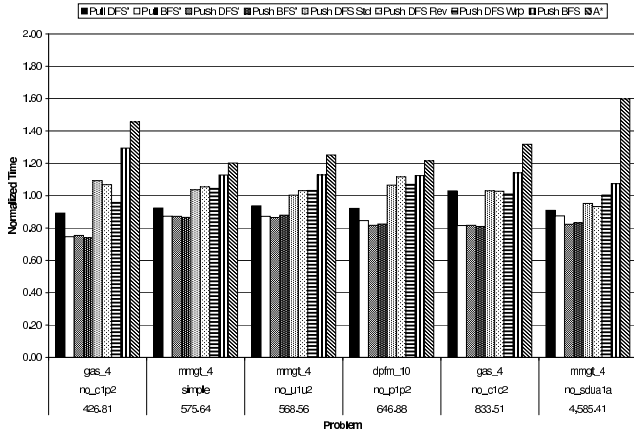


Figure 7. Conclusive problems

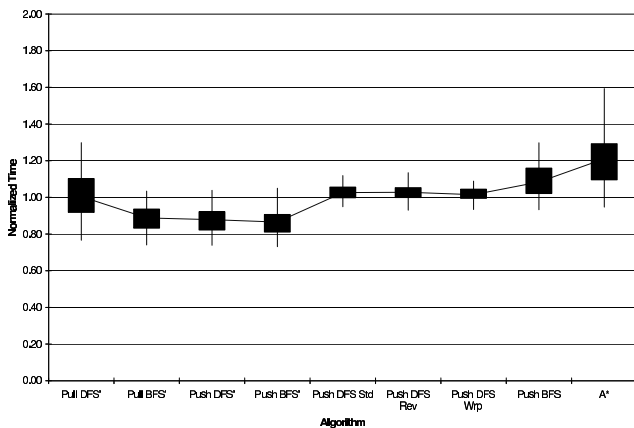


Figure 8. Conclusive averages

formance on conclusive problems.

Figure 7 shows the running times for the algorithms over six of the largest conclusive problems and Figure 8 shows the averages over the 41 conclusive problems that ran longer than 10 seconds for each algorithm.

In these two figures, we see that there is less variance among the different algorithms than among the inconclusive problems. Since these problems are conclusive, all of the algorithms must consider all reachable node-tuples and they all do approximately the same amount of work. The algorithms that place nodes on the worklist all perform better than those that use node-tuples. Since the node-space algorithms maintain less detail, it is understandable that they require less time. The A\* algorithm had the worst time performance, since the operations on the priority queue are  $\mathcal{O}(\log n)$  compared to  $\mathcal{O}(1)$  for the other algorithms.

Of the four algorithms that place nodes on the worklist, it appears that the pushes performed better than the pulls. The pushes probably performed better because they had a simpler implementation. In the pull algorithms, each node is responsible for remembering which tuples it has processed

Algorithms		$t$	$P(T \geq t)$	$\sigma$
Pull DFS'	Push BFS'	8.79	0.00	0.02
Pull BFS'	Push BFS'	5.59	0.00	0.00
Push DFS'	Push BFS'	3.38	0.00	0.00

Table 3. t-tests for conclusive problems

on each of its parents. In the push algorithms, each node only needs to keep track of tuples it has propagated to the children. The more complicated bookkeeping of the pull algorithms probably accounts for the performance difference. Additionally, the BFS's performed better than the DFS's. In examining the behavior of the two worklists, we discovered that once a node was inserted onto the worklist, on average it remained there longer with a BFS' algorithm than with a DFS' algorithm. This means that a node has more time to "accumulate work" and, thus, more work is done each time a node is removed from the worklist with a BFS' algorithm, so these algorithms have fewer worklist operations.

It appears that Push BFS' is the best algorithm on the conclusive problems. To confirm this, we performed matched pair t-tests comparing Push BFS' to the other algorithms that put nodes on the worklist. For each pair of algorithms, we tested the null hypothesis that Push BFS' is not faster, to the alternative hypothesis that it is faster. The results of these tests, shown in Table 3, allow us to accept our hypothesis that Push BFS' is the best of these four algorithms and would be our choice for doing analysis in maintenance mode.

## 6.4. Threats to validity

While we did a careful study of these nine algorithms over the suite of test problems, there are serious threats to the validity of this research. First, although our test suite included 220 different problems, many are derived from a small base set and most were developed to evaluate verification systems. As a result, they may not reflect the types of problems that will occur in real world industrial problems. The Chiron user interface system, on the other hand, is not a contrived problem. Since the performance of our algorithms on Chiron was similar to the performance on the other problems, we have reason to believe that we will see comparable performance on real world problems.

Another issue is the metric we used for judging the usefulness of the path for the fault finding mode. For our study, we used the length of the path as a measure of usefulness. While it is clear that long paths will be of limited use in finding a fault, it is not clear that shortest paths will be the most useful. Unfortunately, we do not have any better metric for measuring the usefulness of a path, although we feel that other metrics need to be developed and investigated.

Finally, we have some concern over the validity of our

inconclusive data. The majority of our inconclusive problems do not contain any faults. They are inconclusive due to the removal of some feasibility constraints that were used to obtain conclusive results. Our intuition was that these inconclusive problems have many paths through the TFG that would result in a violating tuple. We would expect that for a real program with a real fault, that this percentage would be low. That is only a limited number of paths would exercise the fault. For a given problem, we wanted to determine the percentage of paths through the TFG that lead to a violating tuple. Since there can be an infinite number of paths through a TFG, we restricted our interest to only simple paths with no repeating node-tuples. If this percentage is high, then it could indicate that our Push DFS searches performed well because many paths lead to violations. Unfortunately, counting the number of paths is a #P-complete problem, making the problem at least as hard as satisfiability [21]. To get a sense of this percentage, however, we enumerated all the paths of a small example that contains a real fault but has less than 40 node-tuples. There is 1 simple violating path and there are 27 simple non-violating paths. However, in a set of 10,000 random walks through this space, none of the non-violating paths was found. This is because the 1 violating path was less than half the length of the shortest non-violating path, and as a result the algorithm was more likely to find the short violating path. On another example, with around 3,000 node-tuples, we could not enumerate all the paths due to a lack of computing resources. Surprisingly, however, the first 3,000,000 paths found were all non-violating. Clearly, more work needs to be done to evaluate this threat. Since one experiment found lots of violating paths and the second found none, it is hard to see what conclusions can be drawn.

## 7. Related work

Much work has been done in comparing different data flow algorithms, especially algorithms for performing points-to analysis [11, 14]. The problem of comparing verification-based algorithms or of finding counter examples through a data flow analysis problem does not seem to have been studied.

Some work has been done on counter example generation for testing [1, 3]. In [9], model checking is used to create counter examples for the inverse of a property. This returns traces that are paths over which the property holds. These traces are then used to select tests for the system. In this work, two different model checkers, SMV [16] and SPIN [12] were used to create several counter examples for each property. For this study, SMV, which uses a BFS algorithm, generated a large set of short traces, while SPIN, which uses a DFS algorithm, took less time and generated a small set of long traces. A more direct comparison of SMV

and SPIN was made in [2] with comparable results.

Chan et al showed that different algorithms can effect the performance of symbolic model checking [4]. They modified SMV so that it could search either forwards or backwards. In their experiments on a software system, the backwards search worked better. Others, however, have reported that forward search works better on hardware systems [13]. In our work, we do a direct comparison of algorithms to determine which is most useful at which point in time.

## 8. Conclusions

We believe that there are three different modes that an analyst goes through while studying a system, exploratory mode, fault finding mode, and maintenance mode. Of course, in practice, an analyst may move from any mode to another or even be in several modes concurrently at the same for the same system. We believe, however, that for a given property, an analyst has a good sense of what the current mode is, and has corresponding expectations. In each mode an analyst would be best served by an algorithm optimized to meet these expectations.

When in exploratory mode, the goal is to get fast results for inconclusive problems. Push DFS Wrp would be the algorithm of choice here because it was the fastest algorithm on average on our test suite. In fault finding mode, inconclusive results are still expected, but the counter examples should be short. In our experiments, A\* returned the shortest paths in the shortest time. Finally, in maintenance mode, conclusive results are expected. The algorithm that performed best on conclusive problems was Push BFS'. Thus, it appears that different algorithms are best suited for each of these three modes.

We believe that having a separate exploratory and fault finding mode is important. In our experimentation, we have often found that the correct specification of properties is a difficult problem. In exploratory mode, faults with property specification can sometimes be detected. We believe that using an algorithm that can identify faults quickly can aid the correction of property specification errors. Using a more expensive shortest path algorithm to find a path for an incorrect property is wasted time. More experimentation is needed, though, to determine what information is most useful to users in detecting incorrectly specified properties.

Even though our experiments were done using FLAVERS on Ada programs, we believe our results have broader applicability. For example, we would expect similar performance results for programs written in other languages [18]. Additionally, we believe our results have broader applicability to other finite state verification approaches. Two studies [2, 9] reported results that are consistent with our results. From this, we expect that other verification tools will have similar performance

profiles based on the type of search they use. Designers of these tools might want to consider providing users with alternative algorithms so they can make a choice based on their expectations.

We plan to continue this work and conduct experiments over a wider range of problems and algorithms. In particular, the problem of counter example selection requires additional investigation. While the A\* algorithm was an improvement over Push BFS, other heuristics should be considered. A\* examined the state of task automata, but perhaps the state of the property or other feasibility constraints should be used to further improve performance. Although path length is an important metric, we suspect that more powerful and interesting metrics are needed in this area. For example, the analyst might want to have some input in the path selection process, perhaps by providing hints as to what portions of the system should be explored or avoided. We believe that feasibility constraints could be used to support this guidance and plan to investigate this issue further.

## 9. Acknowledgements

We would like to thank Kirk J. Macolini, whose Masters' Project was the inspiration for this work.

This research was partially supported by the Air Force Research Laboratory/IFTD and the Defense Advanced Research Projects Agency under Contract F30602-97-2-0032 by the National Science Foundation under Grant CCR-9708184 and by IBM Faculty Partnership Awards dated 5/21/99 and 6/20/2000. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied of the Defense Advanced Research Projects Agency, the Air Force Research Laboratory/IFTD, the U.S. Government, of the National Science Foundation, or of IBM.

## References

- [1] P. E. Ammann, P. E. Black, , and W. Majurski. Using model checking to generate tests from specifications. In *Proc. of the Second IEEE Int. Conf. on Formal Eng. Methods*, pages 46–54, Dec. 1998.
- [2] R. Bharadwaj and C. L. Heitmeyer. Model checking complete requirements specifications using abstraction. *Automated Soft. Eng.*, 6(1):37–68, Jan. 1999.
- [3] J. Callahan, F. Schneider, and S. Easterbrook. Automated software testing using model-checking. In *Proc. of the Second SPIN Workshop*, Aug. 1996.
- [4] W. Chan, R. J. Anderson, P. Beame, and D. Notkin. Improving efficiency of symbolic model checking for state-based system requirements. In *Proc. of the 1998 Int. Symp. on Soft. Testing and Analysis*, pages 102–112, Mar. 1998.
- [5] R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality of A\*. *J. of the Assoc. of Computing Machinery*, 32(3):505–536, July 1985.
- [6] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proc. of the Second ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 62–75, Dec. 1994.
- [7] R. Ford. Concurrent algorithms for real-time memory management. *IEEE Software*, pages 10–23, Sept. 1988.
- [8] K. Forester, C. MacFarlane, M. Cameron, and G. Bolcer. Chiron-1 user manual. Arcadia Document UCI-93-07, U. of California, Irvine, Sept. 1993.
- [9] A. Gargantini and C. Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. of the Seventh European Soft. Eng. Conf. held jointly with the Seventh ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 146–162, Sept. 1999.
- [10] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, Mar. 1985.
- [11] M. Hind and A. Pioli. Which pointer analysis should I use? In *Proc. of the 2000 Int. Symp. on Soft. Testing and Analysis*, pages 113–123, Aug. 2000.
- [12] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, Inc., 1991.
- [13] H. Iwashita, T. Nakata, and F. Hirose. CTL model checking based on forward state traversal. In *Proc. of the 1996 IEEE/ACM Int. Conf. on Computer-Aided Design*, pages 82–86, Nov. 1996.
- [14] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. of the Seventh European Soft. Eng. Conf. held jointly with the Seventh ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 199–215, Sept. 1999.
- [15] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. *Acta Infomatica*, 28:121–163, 1990.
- [16] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.
- [17] 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 Sixth ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 24–34, Nov. 1998.
- [18] G. Naumovich, G. S. Avrunin, and L. A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proc. of the 22<sup>nd</sup> Int. Conf. on Soft. Eng.*, pages 399–410, May 1999.
- [19] G. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proc. of the Fourth ACM SIGSOFT Symp. on the Found. of Soft. Eng.*, pages 93–105, Oct. 1996.
- [20] G. Naumovich, L. A. Clarke, and L. J. Osterweil. Efficient composite data flow analysis applied to concurrent programs. In *Proc. of the ACM SIGPLAN-SIGSOFT Workshop on Prog. Analysis For Soft. Tools and Eng.*, pages 51–58, June 1998.
- [21] C. H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994.