Using Partial Order Techniques to Improve Performance of Data Flow Analysis Based Verification*

Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh

Laboratory for Advanced Software Engineering Research
Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003
{naumovic, clarke, jcobleig}@cs.umass.edu

Abstract

Partial order optimization techniques for distributed systems improve the performance of finite state verification approaches by avoiding redundant exploration of some portions of the state space. Previously, such techniques have been applied in the context of model checking approaches. In this paper we propose a partial order optimization of the program model used by FLAVERS, a data flow based finite state verification approach for checking user-specified properties of distributed software. We demonstrate experimentally that this optimization often leads to significant reductions in the run time of the analysis algorithm of FLAVERS. On average, for those cases where this optimization could be applied, we observed a speedup of 21%. For one of the cases, the optimization resulted in an analysis speedup of 91%.

*This research was partially supported by the Defense Advanced Research Projects Agency and the Air Force Research Laboratory/IFTD under agreement F30602-97-2-0032, and by the National Science Foundation under Grant CCR-9708184. The views, findings, and conclusions presented here 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, or the U.S. Government.

Appears in Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, Software Engineering Notes, Vol. 24.5, pp. 57 – 65, September 1999.

1. Introduction

Finite state verification techniques automatically check that a software system conforms to a behavior specification or *property*. Such techniques are becoming extremely important with the proliferation of distributed systems. Distributed systems are more difficult to understand and reason about than sequential ones because of the potential non-deterministic interleaving of execution sequences from different threads of control, or *tasks*. While testing demonstrates the actual behavior of a system on selected test cases, distributed systems may not even produce the same results when re-executed with these same test cases. Finite state verification techniques, however, are capable of verifying a restricted, but interesting, class of properties for all possible executions of a program for all possible test cases. Unfortunately, in practice, finite state verification tools often require significant computing resources, and so there is a need for optimizations that improve the performance of such techniques.

Many finite state verification techniques reason about systems in terms of sequences of events, where an *event* corresponds to some recognizable behavior of the system, such as a method call, a task interaction, or an assignment to a variable. One popular method for improving the performance of finite state verification techniques, without compromising the results of their analyses, is *partial order* optimization. Such an optimization is based on the observation that most representations of distributed systems, used in analysis, model the execution of the system as a total order between occurrences of events. This means that multiple executions exist that differ from each other only by the relative order of appearance of events occurring in different tasks. In many cases, these differences are not important for checking the property of interest. In such cases, partial order reduction techniques choose and reason about a single representative ordering.

Necessarily, whether or not two interleavings can be considered equivalent depends on the property being checked. Thus, all partial order methods are defined for specific

Wolper [6], Valmari [13], and Katz and Peled [8] use partial orders for verifying safety properties. Peled [12] proposed a partial order method that allows checking stuttering-close Buchi automata properties.

Partial order methods have been shown to be successful in improving the performance of model checkers. One case study [1] showed that for many situations these techniques significantly improve both time and space requirements of the SPIN [7] model checker, thus enabling analysis of bigger problems. Godefroid, Peled, and Staskauskas [5] describe the design of a partial-order algorithm for a formal validation tool used for verification of several subsystems within Lucent Technologies 5ESS telephone switching system. Godefroid [3] also uses partial order techniques in VeriSoft, a verification tool for distributed systems implemented in C or C++. VeriSoft was demonstrated to be effective in verifying an example of the Lucent Technologies' Heart-Beat Monitor of a telephone switch system [4].

To date, partial order reduction techniques have been applied in the context of model checking approaches. Such approaches enumerate all possible states of the system and reason about these states. In this paper we propose a partial order reduction for FLAVERS, a finite verification approach based on data flow analysis [2]. Although FLAVERS is a general approach that to date has been applied to Ada [2], Java [10], C++, and Jovial programs, as well as an architecture description language [11], in this paper we use Ada as our example language. The program model that FLAVERS uses does not enumerate all possible states of the concurrent system under analysis, but instead it uses a special type of edge to represent possible interleavings between events in different processes. The optimization that we propose in this paper uses partial orders to eliminate some of these edges, thereby improving efficiency of the FLAVERS analysis.

We evaluated the benefits of this optimization on a number of small, distributed programs. As expected, it was relatively easy to determine if the partial order optimization technique was applicable to a problem. For the 92 problems that we considered, the optimization was applicable to 35 of them. On average, for all applicable cases, the speedup of the FLAVERS analysis due to the use of this optimization was 21%. For one case, the optimization resulted in an analysis speedup of 91%.

In the next section we present a high-level overview of FLAVERS and give a detailed description of the system model that FLAVERS uses. Section 3 describes the intuition behind the partial order optimization approach that allows us to remove a significant number of edges from the FLAVERS program model. Section 4 presents this optimization in detail. In Section 5 we present our experimental results. We conclude with observations and future research directions.

2.1. Overview

FLAVERS (**FL**ow **A**nalysis for **VER**ification of **S**ystems) compactly represents a concurrent software system as a *Trace Flow Graph (TFG)* and uses an efficient fixed point data flow algorithm to determine if the behavior described by the TFG is consistent with a user-specified, event-based, safety property. The results of the FLAVERS analysis are conservative; in other words, the technique never claims that a property is verified when it is not. For efficiency reasons, similar to other finite state verification techniques, the TFG model used by FLAVERS over-approximates the potential executable sequences of events associated with the program. This leads to the possibility of *spurious results*, where FLAVERS reports a property violation when there is in fact no real executable behavior of the system that would violate that property. If FLAVERS detects a property violation, it provides the user with example paths that illustrate this property violation. By examining such paths users can often determine if the result of the analysis is spurious or not.

FLAVERS provides a flexible way for improving the precision of the analysis. Analysts do this by adding *feasibility constraints*, which specify additional semantic information about the system and which are used to limit the exploration of the TFG to only those paths that satisfy these feasibility constraints. If the constraints are well chosen by the user, infeasible executions are eliminated and the subsequent analysis run will either verify the property or expose a counter example that corresponds to real executable behavior and, thus, exposes a bug in the system. FLAVERS provides automated support for creating several classes of feasibility constraints, such as constraints that model boolean, counter, or enumerated variables or model control flow through a specific thread of control.

Unfortunately, the use of constraints may lead to larger and more complex data flow problems, since the worst-case complexity of the FLAVERS analysis algorithm is $\mathcal{O}(N^2S)$, where N is the number of nodes in the TFG and S is the product of the number of states in the property and all constraints. Thus, if many constraints are used, the analysis algorithm may have to deal with vast amounts of data. This led us to search for techniques for improving space and time characteristics of FLAVERS. In order to understand the technique proposed in this paper, in the remainder of this section we describe the TFG model in detail and then briefly illustrate the FLAVERS analysis on this model.

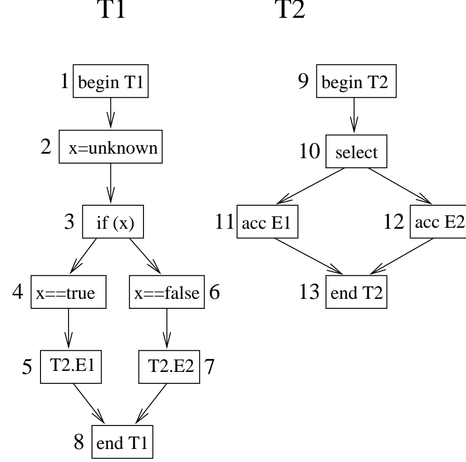
2.2. TFG Model of Ada Programs

The TFG for an Ada program is based on the control flow graphs (CFGs) for all tasks. Additional nodes and edges are added to the TFG to represent intertask communications. Specifically, if the code region represented by node n in one task contains a synchronization statement that can

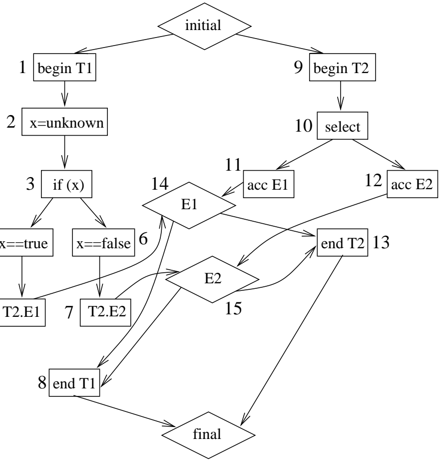
```
task body T1 is
begin
  read(x);
  if (x) then
    T2.E1;
  else
    T2.E2;
  end if;
end T1;

task body T2 is
begin
  select
    accept E1;
  or
    accept E2;
  end select;
end T2;
```

(a) Code

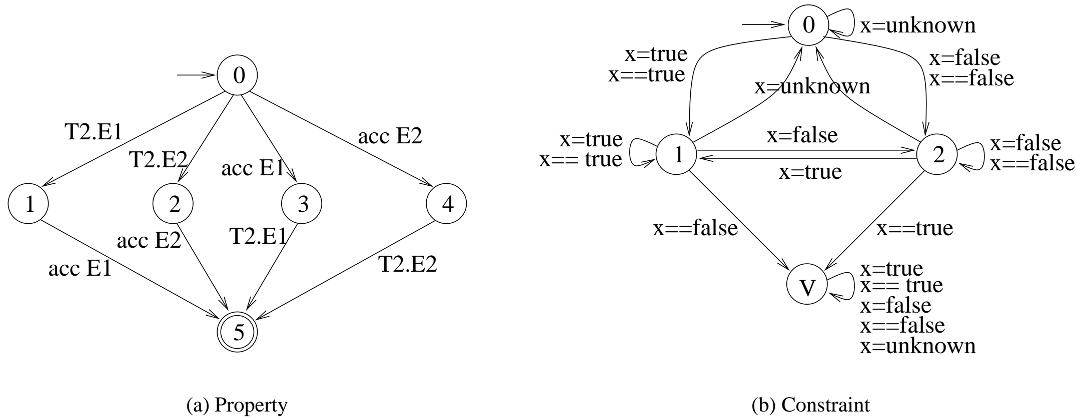


(b) CFGs



(c) TFG

Figure 1: An example



(a) Property

(b) Constraint

Figure 2: Property and constraint example

correspond to one represented by node m in another task, a new node is added with incoming edges from n and m and outgoing edges to all successors of n and m. This is illustrated in Figure 1(c). A unique *initial* node that has no incoming edges and has outgoing edges to the start nodes of all CFGs and a unique *final* node that has no outgoing edges and has incoming edges from the end nodes of all CFGs are added to the TFG.

Formally, a TFG is a labeled directed graph $(N, E, n_{initial}, n_{final}, \Sigma, label)$, where N is the set of nodes, $E \subseteq N \times N$ is the set of edges, $n_{initial}, n_{final} \in N$ are unique initial and final nodes, Σ is the set of labels on the nodes of the graph, and *label* is a mapping from nodes to labels in Σ. The set of all nodes from the CFGs for all tasks forms the set of *local* TFG nodes. All other nodes, except the initial and the final nodes, represent intertask communications and thus are called *communication* nodes.

Let T be the set of tasks in the program. In the TFG for an Ada program, nodes may belong to one (local nodes), two (communication nodes), or no (initial and final nodes) tasks. We use function $task : N \rightarrow 2^T$ to associate with each TFG node the set of tasks it belongs to.

Figure 1(a) shows a program that consists of two communicating Ada tasks and Figure 1(b) shows the CFGs for these two tasks with nodes labeled with the corresponding Ada program statements. Since we will be interested in reasoning about values of variable `x`, information about this variable is included in the CFG. The two nodes labeled `x==true` and `x==false` represent the value of variable `x` being `true` on the first branch of the `if` statement and `false` on the other branch, respectively. The node labeled `x=unknown` corresponds to the `read(x)` statement and means that an unknown value is assigned to variable `x`.

Figure 1(c) gives the TFG for this Ada program. The local nodes in this TFG have the same ID numbers associated with them as the corresponding nodes in the CFGs. The diamond-shaped nodes are the initial and the final nodes of this TFG and the two communication nodes, labeled `E1`

between the tasks at entry call `T2.E1`, and node labeled `E2` represents the communication at entry call `T2.E2`.

FLAVERS represents all properties and constraints as finite state automata (FSA). Transitions in these FSAs must be in Σ. The FLAVERS analysis algorithm propagates the states of these automata throughout the TFG. Let $\mathcal{A}$ be the set containing the property and all constraint automata used in the analysis. Let A be any automaton, either the property or a constraint. Let Σ_A be the alphabet of this automaton; that is, all events used in the transitions in this automaton. Define alphabet $\Sigma_{\texttt{aut}}$ to contain all events in the property and constraint automata: $\Sigma_{\texttt{aut}} = \bigcup_{A \in \mathcal{A}} \Sigma_A \subseteq \Sigma$.

Figure 2 shows a sample property and constraint that can be created for the example in Figure 1(a). The alphabet of the property in Figure 2(a) is {`T2.E1`, `T2.E2`, `acc E1`, `acc E2`}. This property specifies that there are only four legal sequences of these events, each sequence being represented by a pair of transitions from the initial state 0 to the accept state 5 of the property. Note that the property automaton does not show illegal transitions. For example, the transition on event `T2.E2` is not allowed when the property is in state 1, so if this event is encountered while the property is in this state, it means that the property is violated. The constraint in Figure 2(b) models the behaviors of boolean variable `x` in task `T1`. The alphabet of this constraint is {`x=unknown`, `x=true`, `x==true`, `x=false`, `x==false` }. The start state 0 of this constraint represents `x` having value either `true` or `false`. State 1 represents this variable having value `true` and state 2 represents this variable having value `false`. Transitions between the states of this constraint are based on the assignments to variable `x` and on events that appear on the branches of conditional statements using this variable as a predicate. For example, event `x==true` represents an assertion that the value of `x` is `true`. State `v` is the so-called *violation* state. If a constraint enters its violation state during the traversal of a node by FLAVERS, this means that the currently traversed path does not conform to the set of behaviors described by this constraint and thus this path is spurious.

Events from the TFG alphabet Σ that are not present in $\Sigma_{\texttt{aut}}$ are not used by the property and constraints, and so they can be replaced in the TFG with a single event τ. (In the following we assume that $\tau \in \Sigma$.) Figure 3 shows the TFG obtained from the TFG in Figure 1(c) by substituting events that are not in the alphabets of the property and constraint from Figure 2 with τ.

Theoretically, all τ-labeled nodes can be removed from the TFG in a safe manner, with the results of the FLAVERS analysis on the reduced TFG being the same as the results on the original TFG. When a τ-labeled node is removed, an edge is constructed from each predecessor of this node to each successor of this node. For nodes with multiple successors and predecessors this may lead to a quadratic blow-up in the number of control edges in the graph. We use a

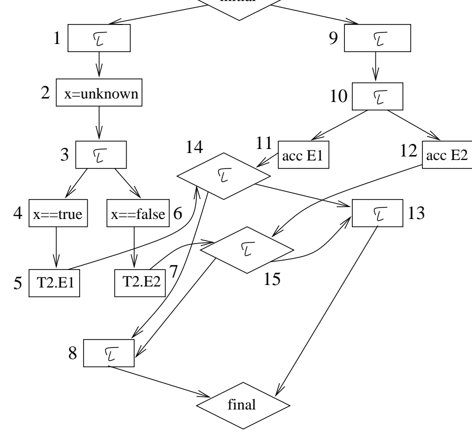


Figure 3: The TFG after replacing some events with τ

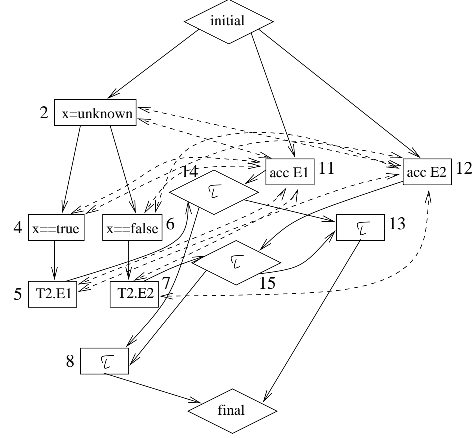


Figure 4: The TFG after removal of some τ-labeled nodes

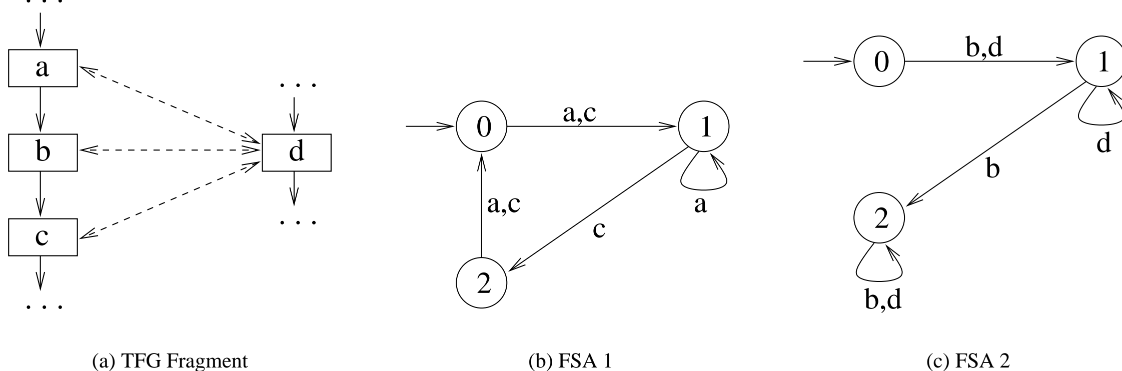
simple heuristic that removes τ-labeled nodes only where this does not lead to an increase in the number of edges in the TFG. In addition, we do not remove a τ-labeled node n in the following two cases:

- n is a communication node,
- removal of n would result in two nodes p and s being connected by a control edge, where p is either the initial or a communication node and s is either a communication or the final node[1].

Figure 4 shows the TFG from Figure 3 after the τ-labeled nodes are removed as described. The dashed edges between the nodes in Figure 4 are explained below.

In addition to edges that represent control flow within a single task, TFGs include edges that represent that execution of a statement in one task *May Immediately Precede* execution of a statement from another task. These edges are referred to as *MIP* edges. For Ada programs, we create

[1] These are requirements of the algorithm [9] that is used for computing MIP edges described in the next paragraph.



(a) TFG Fragment (b) FSA 1 (c) FSA 2

Figure 5: A TFG fragment and two FSAs for an intuitive explanation of the optimization

a MIP edge from node n to node m and a MIP edge from node m to node n if there is a possibility that during some execution of the system regions of code that correspond to these two nodes may execute in parallel and if neither of nodes m and n is a τ-labeled node[2]. We use an efficient data flow algorithm [9] that computes a conservative estimate of such node pairs (n, m). Figure 4 shows the TFG where the MIP edges are shown as dashed lines. Note that each such line represents two edges, going in opposite directions.

2.3. FLAVERS Analysis

Analysis of a property of interest by FLAVERS is based on a data flow algorithm that examines all paths through the TFG that start in the initial node and end in the final node. Events on the nodes along such paths are applied to the property and all constraint automata. If this sequence of events puts the property in an accepting state and each of the constraints in a non-violation state, we say that the property holds on this path. For example, the path (initial, $2, 4, 5, 11, 14, 8$, final) corresponds to the sequence of events (`x=unknown`, `x==true`, `T2.E1`, `acc E1`, τ, τ), which puts the property automaton in the accepting state 5 and the constraint automaton in state 1. Thus, the property holds on this path. If the property holds on all paths through the graph, it holds on all possible executions of the program.

The number of MIP edges in the TFG for a program depends on the synchronization pattern between the tasks in this program and the number of nodes with labels other than τ. In general, the number of MIP edges is quite large, far exceeding the number of control edges. This means that the analysis algorithm has to propagate information through a large number of edges in the graph, which may lead to poor performance. Thus, a reduction in the number of MIP edges would reduce the run time of the analysis algorithm. In addition, such a reduction could improve the precision of the analysis (the number of false negative results), because some paths through the graph that do not correspond to any real executions of the program would be eliminated. Note that such a reduction has to be *safe* with respect to a property, which means that the reduction does not eliminate sequences of events that correspond to real program executions that violate the property. In the next two sections we introduce a safe MIP edge reduction based on partial orders.

[2] Traversal of such τ-labeled nodes from different tasks does not affect the property and constraints.

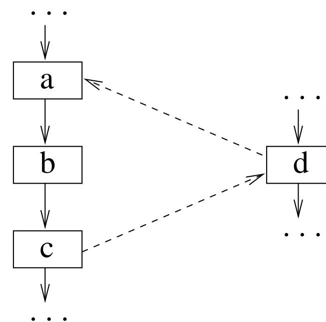


Figure 6: The TFG fragment from Figure 5(a) after removing some MIP edges

3. Intuitive Explanation of the Optimization

In this section we give an intuitive explanation of how partial orders can be used to remove some of the MIP edges from the TFG. Figure 5(a) shows a TFG fragment where three sequentially connected nodes, labeled `a`, `b`, and `c`, represent code that may happen in parallel with code represented by the node, labeled `d`, from some other task. MIP edges are created to represent all possible interleavings of event `d` with events `a`, `b`, and `c`. This yields four possible event interleavings in this fragment: `dabc`, `adbc`, `abdc`, and `abcd`. We assume for simplicity that there are only two tasks in the program and that the first task does not contains event `d` and the second task does not contain events `a`, `b`, and `c`.

Figures 5(b) and 5(c) show two FSAs that we selected to use for the analysis of this TFG. (For this discussion, it does not matter which one of these FSAs is a property and

so $e' = e$.

Σ_{global} is defined simply as $\Sigma \setminus (\Sigma_{local} \cup \{\tau\})$. We call events from Σ_{local} *local* events and events from Σ_{global} *global* events. Also, we refer to nodes labeled with local events *locally-labeled* and to nodes labeled with global events *globally-labeled*. For the purposes of this optimization, we define the initial and the final node of the TFG to be globally-labeled. Note that this separation of the TFG nodes into locally- and globally-labeled is property- and constraint-specific.

In the TFG in Figure 4, events `T2.E1`, `T2.E2`, `acc E1`, and `acc E2` are global, since they are in the alphabet of the property FSA in Figure 2(a) and this FSA contains events from both tasks `T1` and `T2`. Events `x=unknown`, `x==true`, and `x==false` are local, because they are only in the alphabet of the constraint FSA in Figure 2(b) and this FSA only has events from task `T1`. Hence, nodes 2, 4, and 6 in the TFG in Figure 4 are locally-labeled and all other nodes in this TFG are globally labeled.

The idea behind the partial order reduction of the MIP edges is based on this distinction between locally- and globally-labeled nodes. Intuitively, the order of the execution of two locally-labeled nodes that belong to different tasks does not matter for checking of the property of interest, because traversal of these two nodes by the analysis algorithm affects disjoint subsets of the property and constraint automata. Thus, there is no need to represent interleavings of such two locally-labeled nodes by creating MIP edges between them. We never create MIP edges leaving locally-labeled nodes.

To define the reduction algorithm more precisely, we introduce the notion of marked nodes. To compute such nodes, we locate all non-τ labeled nodes s in the TFG such that for a task $t \in task(s)$ there is a path in t from some globally-labeled node p to s, where the only nodes on this path are τ-labeled nodes and the only edges are control edges. (Note that a path can be a single edge if p is s's direct predecessor.) We call such nodes s *marked*. All other nodes are *unmarked*. Since each communication node belongs to two tasks, it is marked if there is a control path in one of these tasks from some globally-labeled node to this communication node, where the only nodes on this path are τ-labeled nodes.

Marked nodes are the only nodes in the TFG that require incoming MIP edges. All incoming MIP edges to unmarked nodes n can be removed, provided that there are MIP edges into marked nodes s that precede n in the control flow, because traversal of all locally-labeled and τ-labeled nodes on the control paths between s and n affects only those FSAs that cannot be affected by traversal of nodes from other tasks.

Once all marked nodes in the TFG are computed, we remove an existing MIP edge from node r to node n if either (1) r is a τ- or locally-labeled node and not a direct prede-

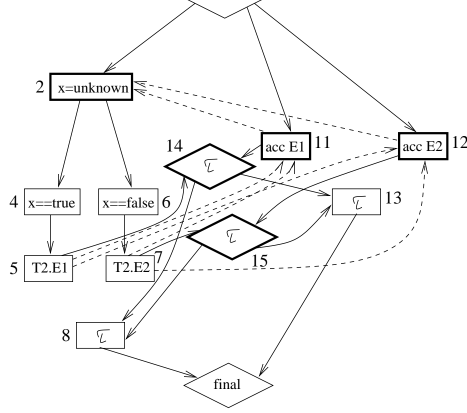


Figure 7: The TFG after the MIP optimization

which one is a constraint.) Note that the alphabet of the FSA in Figure 5(b) is $\{\mathtt{a}, \mathtt{c}\}$ and so it contains events from a single task in the program. Since event `d` is not in the alphabet of this FSA, the transitions in this FSA are not affected by this event, and so for this FSA it does not matter whether `d` precedes `a`, occurs between `a` and `c`, or follows `c`. The alphabet of the FSA in Figure 5(c) is $\{\mathtt{b}, \mathtt{d}\}$ and so transitions taken in this FSA depend on whether event `d` happens before or after event `b`. Thus, our analysis of the alphabets of the two FSAs reveals that preserving all possible interleavings of events `b` and `d` is important, while preserving all possible interleavings of events `a`, `c`, and `d` is not important. This means that only one of two event interleavings `dabc` and `adbc` and only one of two event interleavings `abdc` and `abcd` have to be investigated by FLAVERS. To remove interleavings `adbc` and `abdc` from the TFG fragment in Figure 5(a), we eliminate some of its MIP edges. The resulting fragment is shown in Figure 6.

This idea of eliminating MIP edges to remove the "unnecessary" event interleavings is behind our partial order optimization. Instead of using a naive approach of generating all possible event interleavings and then identifying those that can be removed, we use an efficient algorithm that decides which of the nodes in the TFG do not need incoming and outgoing MIP edges, based on the labels of these nodes. This algorithm is described in the next section.

4. Reduction of MIP Edges

We identify two disjoint subsets, Σ_{local} and Σ_{global}, of the TFG alphabet Σ, where Σ_{local} contains all events e from Σ that satisfy the condition that every automaton containing e in its alphabet contains events from a single task in the TFG. Formally, event $e \in \Sigma_{local}$ if

$$\forall \text{ automaton } A \in \mathcal{A}, e \in \Sigma_A \Rightarrow$$
$$\forall e' \in \Sigma_A \neg \exists t_1, t_2 \in T, t_1 \neq t_2 : \exists n, m \in N : t_1 \in task(n)$$
$$\wedge\, t_2 \in task(m) \wedge label(n) = e \wedge label(m) = e'$$

Note that this condition never holds for labels on communication nodes, because for any communication node n with

node.

Figure 7 shows the TFG for our example after this partial order reduction of MIP edges was performed. The nodes marked by the reduction algorithm have thicker boundaries. Using this optimization we are able to remove 14 of the 20 MIP edges in Figure 4. For example, the MIP edge from node 2 to node 12, present in Figure 4, was removed by the optimization because node 2 is a locally-labeled node that does not have communication nodes as successors. The MIP edge from node 11 to node 6 was removed because node 6 is not marked. (Note that all MIP edges in Figure 7 are unidirectional, while all MIP edges in Figure 4 are bidirectional.)

We proved that this partial order-based reduction is safe. This proof is based on the induction on the number of MIP edges removed by the optimization. We prove that if a removal of a MIP edge satisfying the requirements above eliminates a path in the TFG, one of two conditions hold: (1) there is at least one other path left in the TFG that places the property and constraint FSAs in the same states as the removed path does, or (2) the removed path does not correspond to a real system execution.

5. Experimental Results

We ran an experiment that evaluates the speedup obtained by using this partial order optimization for a number of small concurrent Ada benchmarks. In the following discussion of the results we will refer to the version of FLAVERS that uses this optimization as the optimized version and to the version of FLAVERS that does not use this optimization as the unoptimized version. We used a Pentium II 400 MHz, with 384 Mb of memory, running Windows NT 4.0 with Service Pack 3. The FLAVERS toolset is implemented in Java; we ran our experiments on the JVM supplied with Visual Cafe 2.5a.

In total, we used 16 different programs, ranging from 2 to 16 tasks and from 84 to 750 lines of code, and 41 different properties. Some of these programs are scalable, and so in a number of cases we used several sizes of the same program. This scaling of the programs and also using different sets of constraints and different properties yielded 92 different problems. For each analysis run, we measured the number of τ-, locally-, and globally-labeled nodes in the TFG, as well as the number of MIP edges in the original TFG and the number of MIP edges left after the reduction was performed. We also measured the time taken by the optimization and the subsequent run of the FLAVERS analysis algorithm.

Out of these 92 examples that we ran, in 57 cases the partial order optimization was not applicable, either because the examples had no MIP edges to start with[3] or had no local events. Note that for such cases the partial order op- pass over the TFG, property, and constraints is sufficient to figure out if any TFG nodes are locally-labeled. Here we present the results only for the 35 examples where the TFG contained some MIP edges and local events.

Figure 8 shows the results of running the two versions of FLAVERS on these 35 examples. The first column of the table gives the name of the program used in the example. For some examples we checked multiple properties, which explains the presence of the same program name in multiple rows. Most of these programs are well-known examples from the concurrency literature, such as the dining philosophers and readers-writers examples.

The next three columns list the number of τ-, locally-, and globally-labeled nodes in the TFG. Columns 5-7 list the number of MIP edges, time for adding the necessary MIP edges to the TFG, and the time for the FLAVERS analysis algorithm in the unoptimized version of the algorithm. The last three columns are the same data for the optimized version of the algorithm, where the time for adding the necessary MIP edges to the TFG includes the time for performing the partial order optimization[4].

It can be seen from the table in Figure 8 that in some cases this optimization improves the run time of the FLAVERS analysis algorithm by an order of magnitude. In the best case, we removed 74% of the edges and on another problem we saw a speedup of the analysis of 91%. More often, the improvements are not as striking but still quite significant. Only in 7 examples did the optimization fail to remove any MIP edges. Out of the examples that benefited from the optimization, on average we removed 25% of the edges, with standard deviation .219. This led to an average speedup of the analysis algorithm of 21%, with standard deviation .2578. The extra overhead in performing the optimization is small, less than .2 seconds for all cases, which is small compared to the run time of the analysis algorithm.

We mentioned in Section 2.2 that this partial order optimization has the potential of improving the precision of the analysis. One benefit of this improved precision is that the user may not need to use some of the constraints that would have had to be used with the unoptimized version. (The constraints improve the analysis precision, and some of these improvements may not be necessary if the TFG itself is precise enough.) For each problem where the unoptimized version gave a conclusive result, we also ran the optimized version with a smaller number of constraints. It turned out that for our problem set the precision improvement resulting from the removal of MIP edges was never sufficient to obtain conclusive results with any of the constraints removed.

[3] Although all examples are concurrent, in some of them all threads of control but one contained only τ-labeled nodes, and so no MIP edges were created, according to the simple optimization described in the beginning of Section 4.

[4] Instead of implementing this approach in the way described in Section 4, where some of the existing MIP edges are removed as an optimization, in our implementation we create only those MIP edges that would not be removed by the optimization

System	τ	Local	Global	MIP Edges	MIP Time, s	State Prop, s	MIP Edges	MIP Time, s	State Prop, s
Chiron	45	68	51	236	0.02	1.51	166	0.19	1.37
Chiron	45	68	51	236	0.02	0.71	166	0.18	0.45
Chiron	45	70	51	236	0.02	0.69	166	0.18	0.46
Chiron	45	68	51	236	0.02	1.60	166	0.13	1.04
Chiron	45	68	51	236	0.01	0.70	166	0.19	0.46
Cyclic 2	17	16	9	190	0.01	0.25	50	0.09	0.11
Cyclic 2	17	12	9	116	0.01	0.11	59	0.10	0.09
Cyclic 4	35	16	21	608	0.02	0.91	273	0.10	0.21
Cyclic 4	33	32	21	1552	0.04	70.91	448	0.14	6.59
DPFM 2	8	18	8	136	0.01	0.08	64	0.09	0.08
DPFM 2	8	9	8	80	0.01	0.08	64	0.08	0.07
DPFM 3	11	27	10	216	0.01	0.10	104	0.09	0.09
DPFM 3	12	9	10	96	0.01	0.08	96	0.09	0.09
DPFM 7	23	63	18	536	0.02	2.02	264	0.13	0.92
DPFM 7	24	9	18	160	0.01	0.21	160	0.10	0.21
DPFM 10	32	90	24	776	0.04	66.52	384	0.18	25.22
DPFM 10	33	9	24	208	0.01	0.20	208	0.11	0.20
DPH 2	22	4	16	122	0.01	0.08	110	0.09	0.08
DPH 3	32	4	22	282	0.01	1.80	266	0.09	1.77
MMGT	47	16	78	1312	0.04	3.27	1280	0.15	3.19
TWH-P	5	11	84	4480	0.19	1.91	2933	0.25	0.76
TWH-P	11	7	16	232	0.01	0.09	202	0.09	0.08
TWH-I	5	11	102	6534	0.21	352.81	4644	0.37	221.94
TWH-I	13	7	18	276	0.01	0.09	243	0.09	0.09
RW 2	14	9	12	112	0.01	0.09	96	0.09	0.08
RW 2	15	5	12	88	0.01	0.09	88	0.09	0.08
RW 4	26	9	20	176	0.01	0.16	160	0.10	0.16
RW 4	27	5	20	152	0.01	0.16	152	0.10	0.16
RW 6	38	9	28	240	0.01	0.99	224	0.10	0.99
RW 6	39	5	28	216	0.01	0.92	216	0.10	0.92
RW 8	50	9	36	304	0.01	13.80	288	0.11	13.37
RW 8	51	5	36	280	0.01	9.56	280	0.12	9.57
Ring 2	13	26	20	556	0.02	0.31	388	0.10	0.24
Ring 3	19	39	28	1992	0.05	5.17	1270	0.15	3.25
Ring 4	25	52	36	4356	0.14	193.62	2692	0.29	117.82

Figure 8: Results of the experiment

6. Conclusion

We have shown how a simple optimization of the program model used by FLAVERS can significantly reduce time requirements of the data flow analysis of user-specified properties of concurrent software. This optimization is dependent on the property of interest and the feasibility constraints that the analysis uses. In particular, this optimization tends to work well with analyses that use feasibility constraints modeling variables local to tasks, because labels that represent operations on such variables tend to be local.

As presented in this paper, this optimization is specific to the Ada concurrency model. The only Ada-specific fact that this approach uses, however, is the presence of communication nodes in the TFG. The proposed approach for applying FLAVERS to concurrent Java programs [10] models the Java concurrency without using the Ada-style communication nodes. With some simple modifications, this partial order optimization can be easily extended to this Java-specific program model.

Potentially, the partial order optimization of FLAVERS described in this paper can be further improved. For example, we can use information about which statements may happen in parallel to broaden the definition of local events. Suppose that events `e1` and `e2` are used by the same FSA and appear in different tasks (and thus are global by the definition of this paper) but may never happen in parallel. Then it may be possible to consider these two events local. Another potential improvement is to consider all τ-labeled nodes as locally-labeled nodes. This could lead to a further reduction in the number of MIP edges, although it could have the undesired side-effect of increasing the size of some feasibility constraints necessary for the analysis. We plan to experiment with this trade-off. In addition, we plan to explore a number of directions for further improvements of FLAVERS. For example, variables can be repre-

represented by finite state automata. This would remove the need to create and store variable automata and thus is likely to improve the analysis performance. We expect this and other optimizations to further improve both space and time requirements of the FLAVERS analysis, increasing its applicability to a wider range of concurrent programs.

References

[1] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Transactions on Software Engineering*, 22(3):161–180, Mar. 1996.

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

[3] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 174–186, Jan. 1997.

[4] P. Godefroid, R. S. Hanmer, and L. J. Jagadeesan. Model checking without a model: an analysis of the Heart-Beat Monitor of a telephone switch using VeriSoft. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 124–133, Mar. 1998.

[5] P. Godefroid, D. Peled, and M. Staskauskas. Using partial-order methods in the formal validation of industrial concurrent programs. *IEEE Transactions on Software Engineering*, 22(7):496–507, July 1996.

[6] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the 3rd International Workshop on Computer Aided Verification*, pages 332–342, July 1992.

[7] G. J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall Software Series, 1991.

[8] S. Katz and D. Peled. Verification of distributed programs using representative interleaving sequences. *Distributed Computing*, (6):107–120, 1992.

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

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

L. J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 77–93, Nov. 1997.

[12] D. Peled. Combining partial order reductions with on-the-fly model checking. In *Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, June 1994.

[13] A. Valmari. A stubborn attack on state explosion. In *Proceedings of Computer-Aided Verification (CAV'90)*, pages 156–165, June 1991.