Efficient Composite Data Flow Analysis Applied to Concurrent Programs*

Gleb Naumovich, Lori A. Clarke, and Leon J. Osterweil

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

Abstract

FLAVERS, a tool for verifying properties of concurrent systems, uses composite data flow analysis to incrementally improve the precision of the results of its verifications. Although FLAVERS is one of the few static analysis techniques for concurrent systems that has the potential to handle large scale systems, it sometimes can still be very expensive to use. In this paper we experimentally compare the cost of two versions of this approach for solving composite data flow analysis problems. The first version, product-based, uses the more straightforward approach, and the second, tuple-based, is built around the idea of reducing analysis space requirements at the expense of analysis time. We demonstrate experimentally, by analyzing properties of actual concurrent programs, that the tuple-based version is comparable in time to the product-based version but for large composite data flow problems it requires several orders of magnitude less space.

Keywords

Static analysis, data flow analysis, concurrency

1 Introduction

With the rapid improvement of Web technology, distributed and concurrent systems are becoming increasingly common. Concurrent systems are more difficult to understand and reason about than sequential ones because of their inherent non-determinism. This non-determinism makes testing of such software extremely difficult. One cannot, for example, safely assume that two test runs using the exact same input will necessarily produce the same result. Static analysis approaches provide an important complement to testing approaches, in that they are able to evaluate all potential executable paths for certain classes of faults, and can, therefore, often demonstrate the absence of such faults.

FLAVERS [3, 10] is one such static analysis tool. It uses data flow analysis to prove or disprove application-specific properties of concurrent systems. FLAVERS provides its users with the capability of specifying additional information about the system. This improves precision of the analysis but requires more computational resources.

In this paper we explore an optimization of the FLAVERS approach that greatly reduces space requirements of the tool without sacrificing its time requirements. The initial, intuitive implementation of FLAVERS forms a single structure to represent both the property being checked and the additional information introduced about the system being analyzed. The second version avoids creating this structure, which our experiments show can be enormous in size, by using a tuple representation to keep track of each additional component separately. This reduction in space is paid for by using a more complicated internal representation of the analysis information.

The two versions are compared experimentally. The results of this experiment indicate that not only are space requirements reduced significantly by using the optimization, but in addition no statistically significant execution time penalty is incurred in the process. In addition, we demonstrate that the tuple-based version is better suited for proving multiple properties simultaneously. The technique of improving precision of data flow analyses by identifying spurious executions has been explored before [5, 1]. Holley and Rosen [5] also provide a comparison of several implementations of this technique. The contribution of this paper is in the quantitative evaluation of the two versions of the FLAVERS approach for verifying application specific properties of concurrent programs.

In the next section of this paper we present a high-level overview of the FLAVERS approach, including a description of the internal representations used by the algorithm and an example illustrating the use of additional information to improve analysis precision. After that we present formal definitions of the internal representations used by the two implementations, followed by the description of the product-based

*This work was supported in part by the Air Force Materiel Command, Rome Laboratory, and the Advanced Research Projects Agency under Contract F30602-94-C-0137.

and tuple-based versions. Then our experimental results are presented. We conclude with observations about future research directions.

2 FLAVERS Overview

FLAVERS (**FL**ow **A**nalysis for **VER**ifying **S**pecifications) uses a more compact representation of the software system than most concurrency analysis techniques and uses an efficient fixed point data flow analysis algorithm to determine if the model of the system's behavior is always consistent with the specified intended behavior. FLAVERS provides *conservative* analysis results, in that it never claims that a property is verified when it is not. To be conservative and efficient, it over-approximates the executable behavior of the system. Thus, like most static analysis techniques, FLAVERS may report that a problem may exist when there is in fact no real executable behavior of the system that would cause such a problem. Such a report is known as a *spurious result*. FLAVERS also produces a report that details the path(s) along which all discovered problems might occur. By examining such paths users can often determine if a result is spurious or not.

One of the strengths of FLAVERS is that it also provides a flexible way for identifying spurious executions that can be removed from consideration, thereby making the analysis more precise. This is done by allowing the analysts to introduce additional semantic information, called *constraints*, about the system. The general approach of [5] is used to limit the exploration of the program to only those paths that satisfy the constraints. If these constraints are well chosen, subsequent analysis runs will either verify the property or expose a counter example that corresponds to real executable behavior and, thus, violates the property.

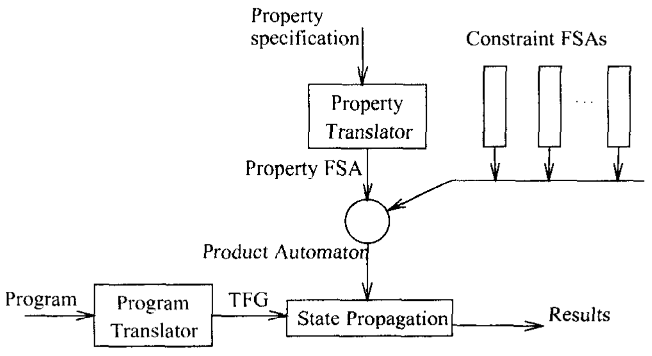

Figure 1: The architecture of FLAVERS

This incremental incorporation of constraints leads to the need to solve increasingly large and complex data flow problems, and this has led us to study techniques for optimizing this approach. To understand the optimization technique that we explored, some more details about the internal representations and algorithm used by FLAVERS are needed.

With FLAVERS, the analyst specifies the property to be verified as a set of sequences of events. Currently FLAVERS uses *Quantified Regular Expressions* language [12] to specify properties. Internally, properties are represented as finite state automata (FSA), called the *property automata*. Similarly each constraint is also represented by a FSA.

Software systems are modeled as *Trace Flow Graphs (TFG's)*. For a sequential program, the TFG is similar to a control flow graph. But for a distributed or concurrent system all possible task interactions must also be represented, as well as all possible interleavings of statements among the tasks. Nodes that represent events that appear in the property or constraint automata must be annotated with those events.

FLAVERS uses data flow analysis to compute whether all system behaviors, as captured by the TFG and constrained by behaviors described by the constraints, are contained in the set of behaviors described by the property automaton. Conceptually, the property automaton and the constraint automata are combined into a *product automaton*, which represents the cross product of the property automaton and all constraint automata. Figure 1 illustrates the architecture of FLAVERS.

During the analysis, the set of reachable product FSA states is propagated along the TFG nodes until a fixed point is reached. Thus, a state, s, is in the annotation set at node n, if and only if there is a path from the TFG start node to n that encounters a sequence of node annotations that drives the FSA to state s when the path reaches n. The activity of deriving these node annotations is represented by the **State Propagation** box in Figure 1. The outcomes of this analysis are divided into three categories of interest: 1) the set annotating the final node of the TFG contains only accepting states of the FSA, indicating that the property holds on **all** executions of the program; 2) the set annotating the final node of the TFG does not contain an accepting state of the FSA, which means that the property holds on **no** executions of the program; and 3) the set annotating the final node of the TFG contains at least one accepting state and at least one non-accepting state of the FSA, which means that the property **may** hold on **some** executions.

In the following we give an example that illustrates how constraints are used and incorporated into the analysis. For simplicity, we use a sequential program in this example, but the general principle of specifying properties and constraints holds for concurrent programs as well.

```
procedure Elevator is
  ButtonPressed : boolean;
begin
  ButtonPressed := GetButtonState;
  if ButtonPressed then
    WaitUntilNoObjectInDoorway;
  end if;
  RecordState;
  if ButtonPressed then
    Car.CloseDoor;
  end if;
end;
```

Figure 2: Code for the elevator example

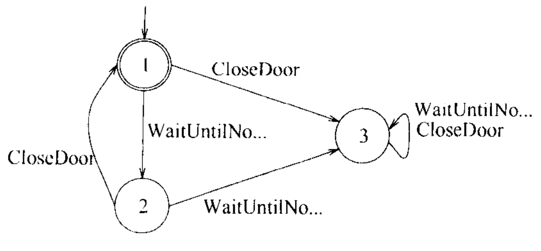

Figure 3: Property FSA for the elevator example

Figure 2 contains pseudocode for an elevator controller. Suppose that the safety property of interest is whether it is possible that the car can close its doors without checking first if there are objects in the doorway. Figure 3 gives an FSA representation of this property. The program is sufficiently simple that it is easy to see that this property holds on all program executions. This is because both the check and closing of the doors are done only if the value of the variable ButtonPressed is true, and if we assume that the procedure RecordState does not change the value of this variable. It is important to note, however, that no information about the values of the program's variables is present in the TFG. This causes FLAVERS to consider some unexecutable paths. For example, the path on which the value of the variable ButtonPressed is assumed to be false in the first `if` statement, and true in the second, appears to violate the property. One example of a constraint automaton that represents the behavior of variable ButtonPressed is shown in Figure 4. Initially the variable is in **unknown** state. The two transitions **Pressed=t** and **Pressed=f** represent the query of whether this variable's value is true and is false, respectively. If the variable has value true, the query of whether its value is false sends this constraint in the violation state.

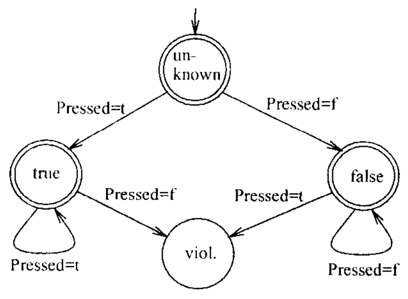

Figure 4: Constraint for the elevator example

Figure 5 shows the TFG for this program, annotated with the events used by both the property and the constraint. All `if` statement branches in this graph are guarded by the nodes with queries of the value of the variable **ButtonPressed**. Now consider the unexecutable path through this graph involving taking the false branch of the first `if` statement and the true branch of the second `if` statement. At the first node of this path, the initial node of the graph, the constraint automaton is at the start state **unknown**. After passing through

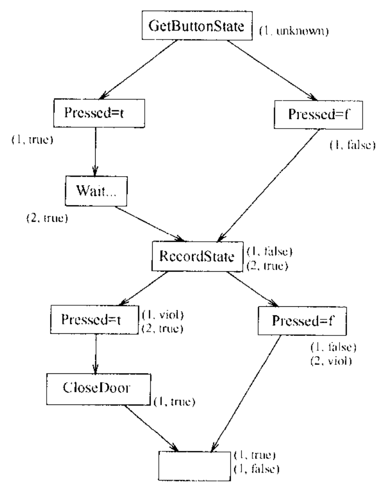

Figure 5: TFG for the elevator example

the successor of the initial node, marked with **Pressed=f**, it takes the corresponding transition to state **false**. After passing through the node labeled with **RecordState**, which does not affect the constraint, this state **false** of the constraint automaton passes through the node marked with **Pressed=t**, at which point the transition to the violation state is taken. Because of this FLAVERS determines that this branch is unexecutable as an extension of the current path.

FLAVERS currently provides automated support for helping users model two specific kinds of constraints, namely variable and task automata. Variable automata, similar to the one in Figure 4, model the execution behavior of scalar variables in the program and task automata model all possible orders of events allowed by the control flow in a single task. In addition, an analyst can construct any arbitrary FSA and use it as a constraint. This approach is very general and allows an analyst to represent the external environment or missing software components [10].

3 Basic Definitions

In this section we give formal definitions for the artifacts that are used in the analysis described in this paper.

A *Trace Flow Graph (TFG)* is a labeled directed graph $T = (N, E, n_{initial}, n_{final}, \Sigma_T, L)$, where N is a set of graph nodes, E is the set of edges, $n_{initial} \in N, n_{final} \in N$ are unique *initial* and *final* nodes, Σ_T is an alphabet of event labels associated with the graph, and $L : N \to \Sigma_T$ is a function that labels the nodes of the graph with event labels drawn from the alphabet. Synchronizations between different tasks are represented explicitly in the TFG, making use of interleaving semantics for the language in which the program is written.

A *Deterministic Finite State Automaton* (or just *automaton* or *FSA*) is a tuple $(S, \delta, s, A, \Sigma)$, where S is a set of states $\{s_1, s_2, ..., s_m\}$, Σ is the finite alphabet of events associated with transitions in the automaton, δ is a total transition function $S \times \Sigma \rightarrow S$, s is a unique *start* state, and A is a set of *accepting* states $\{a_1, a_2, ..., a_p\}$.

A *property automaton* is an FSA $P = (S_P, \delta_P, s_P, A_P, \Sigma_P)$. A *constraint automaton* is an FSA $C = (S_C, \delta_C, s_C, A_C, \Sigma_C)$ with an additional v_C component, called a *violation* state, which is used by the state propagation algorithm to detect that a constraint is violated. For any state $t \in S_C$ and any event $l \in \Sigma_C$, $\delta_C(t, l) = v_C$ if and only if observing event l at state t does not correspond to any legal behavior of the constraint. The violation state is a sink, which means that there are no transitions from this state to any other state in the automaton. Intuitively a constraint specifies a set of desired or expected state transitions, but also explicitly specifies which transitions are not permissible from the current state.

In the following two sections we describe the two approaches to the implementation of the analysis of a single property represented with a property automaton P on the TFG T under k constraints given by constraint automata $C_i, 1 \leq i \leq k$. We require that all events in the alphabets of the property and all of the constraint automata be subsets of the TFG alphabet: $\Sigma_P \subseteq \Sigma_T$ and $\forall 1 \leq i \leq k, \Sigma_{C_i} \subseteq \Sigma_T$.

4 Product-based Analysis

The *product automaton* D for the property automaton P and constraint automata $C_i, 1 \leq i \leq k$ is defined as the tuple $(S_D, \delta_D, s_D, A_D, \Sigma_D, v_D)$, where

- $S_D \subseteq S_P \times S_{C_1} \times ... \times S_{C_k}$
- $\delta_D : S_D \times \Sigma_D \rightarrow S_D$
- $s_D = (s_P, s_{C_1}, ..., s_{C_k})$
- $A_D = \{(a_0, a_1, ..., a_k) | a_0 \in A_P \wedge a_1 \in A_{C_1} \wedge ... \wedge a_k \in A_{C_k}\}$
- $\Sigma_D = \Sigma_P \cup \bigcup_{i=1}^{k} \Sigma_{C_i}$
- v_D is the unique violation state

Note that the set of product automaton states is not necessarily a full cross product of the set of states in the property and the sets of states in all constraint automata. [3] contains a discussion of some techniques that reduce the size of the space of states of the product automaton. One such technique, for example, is merging all product automaton states which have at least one constraint automaton violation state as a subcomponent: if $t = (t_p, t_{C_1}, ..., t_{C_k}) \in S_D$ and $\exists j$ such that $t_{C_j} = v_{C_j}$ then $t = v_D$.

We associate a function f_n over states of the product automaton with each TFG node n. Given a product automaton state s, f_n generates another state $\bar{s}$ obtained from s by taking a transition labeled with the event associated with this TFG node:

$$\forall n \in TFG, \forall s \in S_D : f_n(s) = \bar{s} \in S_D$$
$$\Leftrightarrow \delta_D(s, L(n)) = \bar{s}$$

We generalize functions f_n to introduce a function over sets of product automaton states for each TFG node:

$$\forall n \in TFG, \forall S \subseteq S_D :$$
$$\phi_n(S) = \{f_n(s) | s \in S \wedge s \neq v_D\}$$

Note that the violation state is not propagated past the node for which it was generated by the function ϕ for that node.

The lattice elements for this data flow problem are sets of the product automata states, the join operation is set union $\cup$, and the functional space F is based on all functions ϕ for individual nodes in the TFG.

Once the solution of our data flow problem converges to a *join over all paths* solution [8], we need to look only at the final node of the TFG to determine whether the property holds. We say that a property *holds* on all paths through the program if after all violation states are discarded from the final node of the TFG, only accepting states of the product automaton are present there.

To illustrate the use of the product automaton for improving accuracy we return to the elevator example in Figure 2. The product of the property automaton from Figure 3 and the constraint automaton from Figure 4 appears in Figure 6. Labels on the states of this automaton are pairs, where the

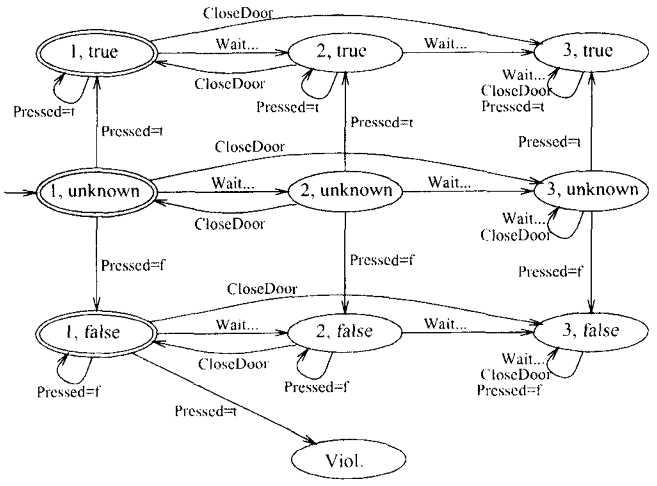

Figure 6: Product automaton for the elevator example

first number corresponds to the state number of the property automaton from Figure 3 and the second label corresponds to the state label of the constraint automaton from Figure 4. This is the product automaton after compaction, since all states in the full cross product with the constraint automaton is in its violation state were fused into a single violation state **Viol**. Note that most of the transitions to the violation state of the product automaton are not shown in the interests of clarity. Consider the unexecutable path through the flow graph where the **true** branch of the second `if` statement is taken after the **false** branch of the first `if` statement. When we trace this path, using it to drive the product automaton in Figure 6, the following sequence of state transitions is observed. From the initial state marked **1, unknown** the transition on event **Pressed=f** is taken to state **1, false**. After passing through the node marked **RecordState**, which does not affect the product automaton, the transition on the next event in the execution trace, **Pressed=t**, leads to the violation state for the product automaton, which signifies that this execution trace corresponds to an infeasible path.

[2] proves convergence of this algorithm to the minimal fixed point and reports the analysis complexity for concurrent systems as $O(|S||N|^2)$. In the worst case a task automaton needs to be constructed for each task. Since the number of states in a task automaton is linear in the number of nodes in the control flow graph for this task, it is obvious that the property automaton can easily be exponential in the number of tasks in the program, which can make the analysis intractable.

5 Tuple-based Analysis

In this section we describe the more space efficient *tuple-based* version. First we introduce the method informally by suggesting the parts of the product-based version that have to be modified, and then we give a formal description of this version.

We begin by observing that most of the states in the full product automaton are not used during the actual analysis. Thus all the memory dedicated to storing these unused states and their transitions is wasted. The tuple-based version overcomes this problem by creating and storing only those combinations of product and constraint automata that are actually used by the analysis.

In this version we traverse all automata separately as we traverse the TFG starting from its $n_{initial}$ node. Initially all property and constraint automata are in their start states. When a node is traversed, its label is matched with the transitions out of the current state of each automaton. If this label is in the alphabet of the property automaton, the corresponding transition is taken, and the property automaton changes state. In the case of a constraint automaton, if a transition on the node label leads to the violation state, this means that the path through the TFG that is being considered is unexecutable, and further traversal down this path will not be continued. During data flow analysis TFG nodes are annotated with sets of tuples, where each tuple consists of a state from the property automaton, and one state from each of the constraint automata. The data flow analysis system must generate a tuple set on exit from each node as a function of the tuple sets found at the exits of each of the node's predecessors. If a generated tuple has at least one of the constraint automata in the violation state, the entire tuple is removed from the analysis as it corresponds to an infeasible execution of the program.

We now present a formal definition of tuple-based analysis. A *tuple* T is a collection of one state from each automaton in the problem.

$$T = (t_P, t_{C_1}, t_{C_2}, ..., t_{C_k}), \text{ where } t_P \in S_P \text{ and}$$
$$\forall 1 \leq i \leq k : t_{C_i} \in S_{C_i}$$

Let $\mathcal{T}$ be the set of all possible tuples:

$$\mathcal{T} = \{(t_P, t_{C_1}, t_{C_2}, ..., t_{C_k}) | t_P \in S_P \wedge \bigwedge_{i=1}^{k} (t_{C_i} \in S_{C_i})\}$$

The *initial* tuple is the tuple $T_0 = (s_P, s_{C_1}, s_{C_2}, ..., s_{C_k})$.

We associate a function f_n over tuples with each TFG node n:

$$\forall T = (t_P, t_{C_1}, t_{C_2}, ..., t_{C_k}) \in \mathcal{T} :$$
$$f_n(T) = (t'_P, t'_{C_1}, t'_{C_2}, ..., t'_{C_k}),$$

where

$$t'_P = \begin{cases} t_P & , L(n) \notin \Sigma_P, \\ \delta_P(t_P, L(n)) & , L(n) \in \Sigma_P \end{cases}$$

$$\forall 1 \leq i \leq k, t'_{C_k} = \begin{cases} t_{C_k} & , L(n) \notin \Sigma_{C_k}, \\ \delta_{C_k}(t_{C_k}, L(n)) & , L(n) \in \Sigma_{C_k} \end{cases}$$

As in the product-based version, we generalize f_n to a function over sets of tuples for each TFG node:

$$\forall n \in N, \forall X \subseteq \mathcal{T} : \phi_n(X) =$$
$$\{f_n(T) = f_n(t_P, t_{C_1}, t_{C_2}, ..., t_{C_k}) |$$
$$T \in X \wedge \bigwedge_{i=1}^{k} (t_{C_i} \neq v_{C_i})\}$$

The lattice elements for this data flow problem are sets of tuples, the join operation, as in product-based analysis, is set union $\cup$, and the functional space is based on the set of ϕ_n functions for all TFG nodes n.

Once the solution of our data flow problem converges to a fixed point, we need to look only at the end node of the TFG to determine whether the property holds. But now the tuple-based version is different from the product-based version. For each tuple in n_{final} we check the possible values of each of its constraint automata to see whether all of these possible states are accepting states. If any constraint automaton is left in a non-accepting state, we remove the entire tuple from n_{final}. We say that a property *holds* on all executions of the program if all tuples remaining in n_{final} contain only accepting states of the property automaton.

This approach of representing information propagated around the flow graph as tuples is reminiscent of *K-Tuple frameworks* from [9]. An important distinction is that each component of a tuple in K-Tuple frameworks corresponds to a special edge kind in a graph. In our approach, an event associated with a TFG node can be present in alphabets of several constraint and property automata and thus components of tuples are not directly tied to disjoint sets of information in the flow graph.

To illustrate the use of the tuple-based version to accuracy improvement we use the same example from Figure 2 that we used for the product-based version. We use the property from Figure 3 and the constraint automaton from Figure 4. Figure 5 shows the TFG for this example annotated with tuples that were formed during the tuple-based analysis. We consider the traversal of the unexecutable path where the **true** branch of the second `if` statement is taken after the **false** branch of the first `if` statement.

A tuple appears next to the node if it is the tuple that was observed at the exit from this node. Note that on the entry to the first flow graph node on the true branch of the second `if` statement, marked **Pressed=t**, the constraint automaton component of the tuple is at state **false**, and so the event **Pressed=t** triggers the transition to the violation state. This means that the resulting tuple **(1, Viol)** will not be propagated beyond the the node marked **Pressed=t**, and so the traversed path is unexecutable. Since in all tuples associated with the final node the property automaton is in accepting state **1**, the property holds on all executions of this program.

The tuple-based version is computationally not much more complex than the product-based version. The only difference arises from the different procedure for checking for constraint violations. In the worst case the complexity of the tuple-based version is $O(k|S||N| + |S||N|^2)$, where $O(|S||N|^2)$ is the complexity of propagating tuples among the nodes in the TFG. $O(k|S||N|)$ is the complexity of computing functions f_n for all nodes in the TFG. It follows from the fact that an application of a function f_n to a single tuple involves computing $k + 1$ transition functions, and at each TFG node we apply its function f_n to at most $|S|$ tuples.

The use of the tuple-based version also has the advantage of being more flexible. For example, it is possible to check several properties at the same time using the tuple-based version, and to simultaneously improve the accuracy of all the analyses through the use of the same set of feasibility constraints. This is done by simply extending the definition of a tuple to include multiple property automata, one for each property to be checked. Note that it would be possible to enable the product-based analysis to check several properties at the same time too, but this would be much more complicated as several kinds of accept states would be needed in order to distinguish among the several property automata used to synthesize the product automaton.

6 Empirical Results

We analyzed program-specific properties of several small concurrent programs. For each program we selected one commonly evaluated property. The specification of the properties is omitted here for lack of space.

Program	Number of tasks	Number of constraints	Number of experiments
Dining philosophers	4	4	11
	8	8	37
Dining philosophers with dictionary	4	4	11
	6	6	22
	8	8	37
Dining philosophers with fork manager	3	5	5
	4	7	8
	5	9	12
Gas station	4	4	11
	5	5	16
	6	6	22
Readers-writers	3	4	11
	4	5	16
	5	6	56
Token ring	4	8	36
	8	12	15
Milner's cyclic scheduler	4	8	152
	8	16	134

Figure 7: Programs used in the experiment

The only kinds of constraints used in this experiment are task and variable automata, since they can be built automatically. For each possible combination of constraints we ran each of the two versions of FLAVERS until the analyses concluded. Depending on which constraints were used, the results of these analyses were either conclusive or inconclusive. In this experiment we do not care which, since we are interested in comparing performance of the two versions in either case. Figure 7 identifies all programs used, giving the number of tasks in the program, the number of constraints available, and the number of experiments that use different combinations of constraints. Note that the number of experiments is less than the total number of possible combinations of constraints since we only include runs where the product-based version did not run out of memory. The combined number of runs of each of the two versions for all programs is 612.

In our experiments we did not use a full product automaton, but rather an automaton produced by applying a standard reduction algorithm [7] and then the heuristics from [3] to the full product automaton. To build this reduced product automaton, the product-based version has to construct the full cross-product of all constraint and property automata for the problem and then reduce it. Thus, it is the size of the unreduced version of the product automaton that limits the problems we can actually solve with the product-based version, but it is the sizes of the reduced product automaton that are actually listed in the tables provided here.

We report the time and space requirements for the analyses as measured by the UNIX **time** command on a DEC Alpha Station 200 4/233 with 128 megabytes of physical memory. The absolute values of time and space requirements may seem staggering at first, but should be easier to accept in light of two considerations. First, as noted earlier, the need to model concurrency adds enormously to the complexity of this problem, as it necessitates explicit representation of all possible interleavings of the events in potentially concurrent tasks. Second, FLAVERS is a prototype analysis tool, whose performance has not yet been fully optimized. In any case, the subject of this paper is not the raw values of these requirements, but rather the reductions achieved by utilization of the tuple-based approach. We are confident that other research will further reduce these raw values.

Figure 8 gives a graphical comparison between space requirements for the two versions. In this figure, product-

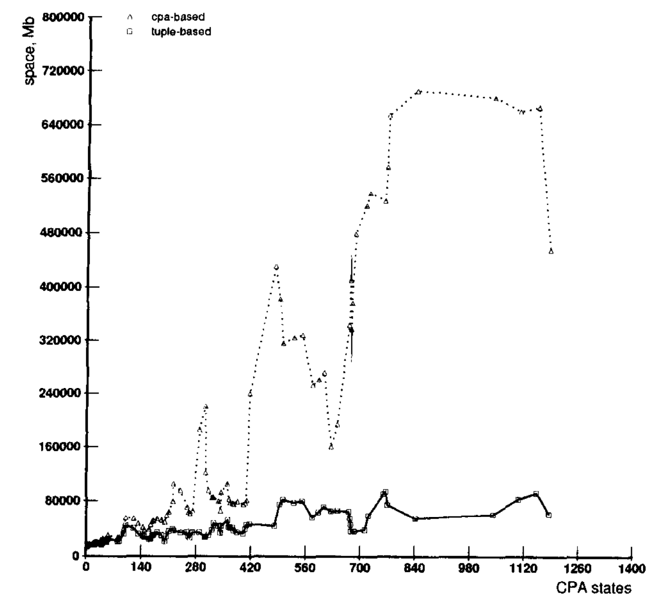

Figure 8: Space requirements comparison

based analysis data points are denoted by triangles and tuple-based analysis data points are denoted by boxes. The graphs for both methods have been smoothed by 3-mean smoothing to improve the viewability of the parts of the graphs that represent small product automata sizes. As is evident from this figure, and as expected, the tuple-based version significantly reduces the space requirements of FLAVERS and hence increases the number and types of analysis problems that can be handled by the tool.

To see where the current limits of the tuple-based version lie, we ran several analyses for the gas station and concurrent writers programs, where we increased the number of constraints used simultaneously. Since the product-based version cannot handle problems of this size, we estimated the number of states in the product automata for these analyses. This comparison is shown in Figure 9. From this figure there appears to be no apparent correlation between the data flow problem size and the space requirements of the tuple-

Estimate of the number of product automaton states	Tuple-based analysis space requirements, Kb
3457	34368
60032	34368
44001	75520
48401	492032
52801	625536
484001	501696

Figure 9: Experiment with problems of larger size

based analysis. We believe that the explanation for this is that as more information is added to the analysis, more paths through the flow graph may be recognized as unexecutable and thus the search space is reduced. This apparent pruning of unexecutable paths does not seem to be a clear function of any obvious parameters of the analysis problem. In general, the tuple-based version seems to handle programs whose product automata would be two to three orders of magnitude larger than what could be stored explicitly.

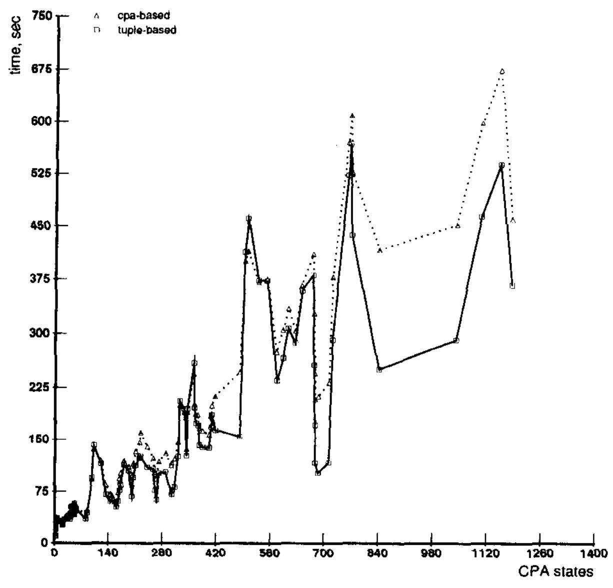

Figure 10: Time requirements comparison

Figure 10 gives a graphical comparison between the speeds of the two versions. From this graph it is clear that improvement in space requirements for tuple-based analysis is not paid for by speed degradation. It seems that the time an analysis takes is of the same order of magnitude for the two versions for all runs. This reflects the fact that the added complexity of tuple propagation through the TFG, as compared to the propagation of states for the product automata, is offset by the time it takes to precompute the product automaton. On average, the tuple-based version has a better time performance, with the mean difference being -15.49 sec.

We performed a number of statistical analyses to estimate the statistical significance of our results. These analyses support the hypothesis that, assuming normal distribution of the sample, for any large-sized sample of Ada programs both space and time requirements for the tuple-based version will be lower than those for the product-based version. In addition, it turns out that the ratio between the space requirements of the product-based version and the space requirements of the tuple-based version grows with increasing the size of the data flow problem. In other words, the tuple-based version scales better than the product-based one. A more complete discussion of the results of this statistical analysis can be found in [11].

7 Conclusions

We have shown how a carefully optimized implementation of the FSA data abstractions can significantly reduce space requirements for composite data flow analyses, while at the same time noticeably improving the speed of these analyses. The experimental results we obtained indicate that the implementation using the tuple-based abstraction can solve much larger data flow problems. This version also ran faster, presumably because the additional propagation work done by the tuple-based version is offset by the work this version saves by not needing to build the potentially enormous product automaton required by the product-based version.

We plan to explore a number of directions for further improving the performance of FLAVERS composite data flow analysis. For example, we shall evaluate representing variables symbolically during state propagation; removing the need to create and store variable automata is likely to improve the analysis performance. We also intend to complement the basic direction of this current work by exploring ways to reduce the size of the TFG's being analyzed. Currently, we model concurrency with TFG's that contain enormous numbers of edges needed to model all possible interleavings of the statements of parallel tasks. Needing to consider all of these edges slows the analysis of such programs considerably. Partial order methods [4, 6, 13] may prove useful in addressing this problem by reducing the need for many of these edges. We expect these and other optimizations of composite analysis to improve both space and time requirements of the analysis, thereby increasing the applicability of this approach to a wider range of both concurrent and sequential programs.

We hope that this work draws attention to the need to explore the balance between the practical complexity of flow algorithms and the representation of data that they use. This paper demonstrates that a shift in this balance can increase the size of problems that can be solved by several orders of magnitude. This alone should serve to greatly broaden the scope of effective applicability of data flow analysis. The significance of this work seems to us to go farther, however. We have already indicated that the composite data flow analysis approach can also be used to solve multiple data flow analysis problems simultaneously. Thus our work shows that use of the tuple-based approach can materially facilitate the solving of multiple problems simultaneously. Although we have conducted this experiment in the context of FLAVERS, we believe the results are more general and can be applied to a range of optimization and analysis problems that utilize data flow analyses.

8 Acknowledgments

We thank Matthew Dwyer, George Avrunin, Daniel Rubenstein, and Tim Chamillard for many helpful discussions about this work. Furthermore, we are grateful for the insightful comments from the anonymous reviewers.

References

[1] R. Bodík, R. Gupta, and M. L. Soffa. Refining data flow information using infeasible paths. In *Proceedings of the 6th European Software Engineering Conference and the 5th ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 361–377, September 1997.

[2] M. Dwyer. *Data Flow Analysis for Verifying Correctness Properties of Concurrent Programs*. PhD thesis, University of Massachussetts, Amherst, 1995.

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

[4] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In *Proceedings of the Third Workshop on Computer Aided Verification*, pages 417–428, July 1991.

[5] L. H. Holley and B. K. Rosen. Qualified data flow problems. *IEEE Transactions on Software Engineering*, SE-7(1):60–78, January 1981.

[6] G. J. Holzmann, P. Godefroid, and D. Pirottin. Coverage preserving reduction strategies for reachability analysis. In *Proceedings of 12th International Conference on Protocol Specification, Testing, and Verification, INWG/IFIP*, Orlando, Fl., June 1992.

[7] J. E. Hopcroft and J. D. Ullman. *Formal Languages and their Relation to Automata*. Addison-Wesley, 1969.

[8] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks. *Acta Informatica*, (28):121–163, 1990.

[9] S. P. Masticola, T. J. Marlowe, and B. G. Ryder. Lattice frameworks for multisource and bidirectional data flow problems. *ACM Transactions on Programming Languages and Systems*, 17(5):777–803, September 1995.

[10] G. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proc. of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 93–105, Oct. 1996.

[11] G. Naumovich, L. A. Clarke, and L. J. Osterweil. Comparing implementation strategies for composite data flow analysis problems. Technical Report UM-CS-1997-043, University of Massachusetts, Amherst, August 1997.

[12] K. M. Olender and L. J. Osterweil. Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[13] A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. Kurshan, editors, *Computer-Aided Verification*, pages 25–41. American Mathematical Society, Providence RI, 1991. Number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science.