FLAVERS: a Finite State Verification Technique for Software Systems*

Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003
(413) 545-2013
{jcobleig, clarke, ljo}@cs.umass.edu

Abstract

Software systems are increasing in size and complexity and, subsequently, are becoming ever more difficult to validate. Finite State Verification (FSV) has been gaining credibility and attention as an alternative to testing and to formal verification approaches based on theorem proving. There has recently been a great deal of excitement about the potential for FSV approaches to prove properties about hardware descriptions but, for the most part, these approaches do not scale adequately to handle the complexity usually found in software. In this paper, we describe an FSV approach that creates a compact and conservative, but imprecise, model of the system being analyzed, and then assists the analyst in adding additional details as guided by previous analysis results. This paper describes this approach and a prototype implementation, called FLAVERS, presents a detailed example, and then provides some experimental results demonstrating scalability.

1 Introduction

Software systems are increasing in size and complexity and, subsequently, are becoming ever more difficult to validate. Testing is the most commonly used technique for validating software, encompassing such diverse approaches as unit testing, integration testing, system testing, regression testing, requirements based testing, and stress testing. Although all these activities serve a valuable purpose, in general, none can assure that a software system will not violate important behavioral properties, such as robustness or safety requirements.

Distributed systems are even more difficult to validate than sequential systems, primarily because of non-determinacy. When there is non-determinacy, the same test case may produce different results on different executions. Thus, testers of distributed systems can not be sure that a software system will continue to produce correct results for previously successful test cases. Moreover, it may also be difficult to reproduce erroneous results with test cases that exposed failures on previous executions.

The limitations of testing have long been recognized [Dij]. Formal verification techniques, based on theorem proving, were originally proposed to counter some of these limitations [Flo67]. Formal verification techniques attempt

*This research was partially supported by the U.S. Department of Defense/Army and the Defense Advance Research Projects Agency under Contract DAAH01-00-C-R231, by the National Science Foundation under Grant CCR-9708184 and by IBM Faculty Partnership Awards. 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. Dept. of Defense, the U. S. Army, the U.S. Government, the National Science Foundation, or of IBM.

to prove that a software system is consistent with a specification of its intended functional behavior. The basic idea is elegant, but demands a considerable amount of mathematical sophistication on the part of the analyst. Although significant progress has been made in providing automated support to assist with formal verification [ORSSC98, CW96], it still requires considerable effort and expertise. As a result, if used, it is usually applied only to small, critical portions of a system.

Finite State Verification (FSV) has been gaining credibility and attention as an alternative to formal verification approaches based on theorem proving. Like theorem-proving based approaches, FSV can be used to verify that all possible executions of a system are consistent with a behavioral specification. FSV uses a finite model of the system and then tries to determine whether that model is consistent with a property specification. If an inconsistency is found, a counter example, representing a trace through the model, can be provided to show how the inconsistency occurs. There are a few key differences between FSV approaches and formal verification based upon theorem proving. One difference is that formal verification can reason about system properties expressed as functions, whereas FSV property specifications are restricted to being notations such as temporal logic predicates or finite state automata. Consequently there are important differences between the reasoning engines employed by the two approaches; FSV reasoning engines typically are guaranteed to terminate, while formal verification theorem provers offer no such guarantees. In addition, FSV seems to require considerably less mathematical expertise from the analyst.

There has recently been a great deal of excitement about the potential for FSV approaches to prove properties about hardware descriptions. FSV systems, such as SMV [BCM$^+$92, McM93] and SPIN [Hol97], have been able to handle reasonably complex descriptions and find erroneous conditions, thereby saving hardware developers considerable expense. To date these approaches require that the system description be represented by a relatively abstract model, which is typically derived with some manual assistance, usually after several attempts and with considerable human ingenuity. There have been some case studies where analysts have successfully represented and verified software systems using these approaches [ABB$^+$96, WVF95] but, for the most part, these approaches do not scale adequately to handle the complexity that is usually found in software systems. Addressing this problem is currently an area of considerable interest and research.

In this paper, we describe our FSV approach for verifying software systems. This approach is based on using efficient data flow analysis techniques to determine if all possible executions of a software system adhere to properties specified as sequences of events. A key difference between our approach and other FSV approaches is that we base our analyses upon a system model that is much smaller, and seems to scale more successfully, than the large state models used by most other FSV techniques. For most FSV techniques, the size of the system model tends to grow exponentially with the size of the system. For our approach this does not appear to be the case. This paper describes our approach and indicates why we believe it will scale more successfully. The paper describes FLAVERS, a prototype system implementing our approach. FLAVERS, **FL**ow **A**nalysis for **VER**ification of **S**ystems, has been developed for application to the analysis of systems written in Ada or Java. The initial experiences we have had with FLAVERS, described in this paper, suggest that there are several reasons why the FLAVERS approach to FSV is well suited for analyzing software. Specifically:

- The system model does not require an enumeration of the entire state space of the system;
- Automated tools can build the model, with little or no intervention from an analyst;
- The model is based on recognizable, user-specified events in the source code, such as method calls, and not on the values of variables;
- The approach is incremental, starting with a small, but imprecise model that the analyst can incrementally sharpen, guided by previous analysis results.

Section 2 provides a high-level overview of the FLAVERS approach. Section 3 carefully describes an example, first for a single task and then for a mutli-threaded system. Section 4 presents some experimental results showing how FLAVERS scales to handle larger systems. Section 5 presents a short description of related work. Finally, the conclusion summarizes the contributions of this approach and discusses future research directions.

2 FLAVERS Overview

FLAVERS is a static analysis approach. Given source code and some behavioral properties of the system, FLAVERS attempts to verify that all possible executions of the system will satisfy these properties. Thus, this verification is done independently of any test data. Testers usually have a goal or requirement in mind when they select a particular test case, and must contrive to select test data whose execution will cause a violation of that requirement. When no violation occurs, the tester is still unsure whether the system will always satisfy the requirement. With FSV, when a requirement is verified, then it is known to be valid for *all* possible test cases.

In this section we describe how FLAVERS expects properties to be represented, the model that it uses to represent the system being analyzed, and the algorithm that FLAVERS uses to determine whether all system executions must satisfy the property. The next section presents an example that illustrates the process of using FLAVERS to verify properties and also demonstrates examples of the artifacts that are created in doing so.

2.1 Representing Properties

The properties that FLAVERS uses for verification must be defined as sequences of events, where an event corresponds to a recognizable executable syntactic entity in the system, such as a method call, a task interaction, or an assignment statement. While this may appear to be overly restrictive, in practice there are many examples of important properties that are expressible this way. For example, it appears that a Mars lander mission recently failed in part because the software system designed to assure a soft landing did not deal correctly with sequences of events pertaining to a particular Boolean variable representing the result of a test for a "bump". The software system assumed that when the lander approached the planet, the "bump" detection variable would be set to false, then the landing gear was to be deployed, and then, when contact with the planet was detected, the "bump" detection variable would toggle from false to true. Setting the variable to true would be the signal for the descent engine to turn off. When the satellite first entered the Martian atmosphere, however, the firing of the lander's retro rockets caused sufficient deceleration to cause a "bump" to be detected, and the "bump" variable was then set to true. The value of the variable was never checked or reset to false prior to deployment of the landing gear, however. Once the landing gear was deployed, checking for the "bump" variable occured. As this variable had previously been set to true because of the retro rocket firing, and had never been reset to false, the condition for shutting down the descent engine was immediately satisfied, and the engine shut down high above the Martian surface, causing the loss of the lander.

Detecting a "bump", testing the value of the "bump" variable, deploying the landing gear, and shutting down the engines are all examples of events that can usually be recognized in well-designed, object-oriented code. In terms of such events, a correct event sequence might be: fire retro rockets, reset bump variable to false, deploy landing gear, detect bump, shut down retro rockets. Thus, it seems possible to represent this event sequence as a property and to use a finite state verification system to determine if faulty landing sequences such as this one could ever occur, thereby jeopardizing the landing on Mars.

FLAVERS requires that a property be represented as a *Finite State Automaton* (FSA). Formally, an FSA is a five-tuple, $F = (S, \delta, s^0, A, \Sigma)$ where S is a finite set of states, Σ is a finite alphabet, $\delta : S \times \Sigma \rightarrow S$ is a total transition function, $s^0 \in S$ is an unique start state, and $A \subseteq S$ is a set of accepting states. In the current implementation of FLAVERS, properties can be represented directly as FSAs or by using an extended regular expression notation, called Quantified Regular Expressions [OO90]. Other specification languages could be used as well, provided that expressions in these languages can be translated into FSA representations. FLAVERS is generally used in one of two ways: to verify that *none* of the possible executions of a system could possibly satisfy a (presumably undersirable) property or to verify that *all* executions must always necessarily satisfy a (presumably desirable) property. Specification of which mode of use is an integral part of the property specification. For simplicity, we assume that every property that we describe in this paper is an *all* property.

2.2 Modeling the System

FLAVERS analysis requires the creation of an abstract graph model of the system being analyzed. The model is a very generic, language-independent representation that depicts all the possible sequences of occurrence of the events of

interest for any execution of the system. As part of the FLAVERS project we have developed tools that automatically produce this model directly from programs written in Ada [DC94, DC99] or Java [NAC99a]. We have also used manual translations to demonstrate the feasibility of building such models for a high-level architectural description language [NACO97] and a process definition language [CCO00].

The FLAVERS system graph models can be directly derived from annotated *Control Flow Graph* (CFG) representations of the system, where annotations are placed on nodes of the CFGs to represent the events that occur during execution of the statements associated with a node. CFGs are used widely in Computer Science, and there is a large body of literature on how to define and generate them (e.g., [ASU86], etc.). Thus, here we assume that the reader is familiar with such a representation and focus our attention instead on the *Trace Flow Graph* (TFG), which is the representation that FLAVERS uses as the basis for its analysis.

A TFG is derived from a collection of annotated CFGs. The TFG is a reduced inlined representation of the CFGs. In the TFG, all method invocations have been replaced by expansions of the procedures that they call, and the resulting graph is then reduced by the removal of all nodes that neither bear event annotations nor affect control flow. In cases where a CFG node is annotated with more than one event, we assume that these events are ordered and replace the node by a sequence of nodes, each annotated with exactly one event. Nodes not otherwise annotated are annotated with a τ event, thereby assuring that each node in the TFG is annotated with exactly one event. Because events tend to occur relatively sparsely in most systems being analyzed, the resulting TFG is usually considerably smaller than the original CFG, despite the size expansion factor inherent in inlining.

Note that a CFG over-approximates the set of possible executions of a system, in the sense that, while any execution of the system can be represented by a path in the CFG, there nevertheless may be paths in the CFG that do not correspond to any actual execution. Correspondingly, while any sequence of events that can occur during an actual execution will be represented by a path in the TFG, nevertheless some event sequences traceable along paths in the TFG may not correspond to executions that could actually occur. Thus, we say that the TFG is an over-approximation, or conservative representation, of the executable event sequences of a system. This is important because it assures that no path on which a property violation occurs will be overlooked. On the other hand, it does leave open the possibility of "false negatives", namely identification of property violations on paths that are not actually executable. Technology to address this problem is incorporated into FLAVERS and is described shortly.

When the system to be analyzed incorporates the use of concurrency, intertask edges and nodes are added to the TFG to represent this concurrency. To analyze concurrent Ada programs, for example, FLAVERS introduces communication nodes and their corresponding edges to represent the rendezvous between two tasks. For all concurrent languages, FLAVERS must represent the potential interleavings of events that may happen during the parallel execution of different threads. Accurately determining all interleavings (up to symbolic execution) is equivalent to creating a reachability graph that enumerates all possible ways in which statements in different threads may execute concurrently. The worst case bound on the size of a system's reachability graph is an exponential function of the number of threads in the system [Tay83]. Because this representation is usually too large to be used in practice, we use a far smaller representation that is computationally more tractable, but is less precise, causing it to over-approximate possible task interleavings. The FLAVERS TFG incorporates a *May Immediately Precede* (MIP) edge between two nodes to represent that execution of the node at the tail of the edge *may* happen immediately before the execution of the node at the head of the edge. MIP edges can be computed relatively efficiently [NA98, NAC99a], but, as noted above, may over-approximate the actual executable interleavings, thereby providing a conservative representation of the sequences of events that could occur during execution of the system.

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, Σ_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.

2.3 Verifying properties

FLAVERS uses an efficient state propagation algorithm to determine whether all potential executions of the system 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.

As noted above, the TFG is a conservative representation of the sequences of events in a system. The results returned by FLAVERS analyses are conservative as well, 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*. Like properties, feasibility constraints are represented as FSAs. A constraint FSA, however, is used to express a particular semantic feature of the system, and is then employed to identify TFG paths that cannot be executed because they violate the semantic feature embodied in the constraint FSA. Constraint FSAs have an added state, known as the constraint violation state, v, that is transitioned to whenever the semantic feature they embody is violated. Thus, for example, a constraint FSA may be created and used to identify paths whose execution would require that a particular variable have the value true and false simultaneously, or a path requiring that the statements of a task be executed in reverse order.

FLAVERS analysts often use constraint FSAs to model the value of program variables that affect the flow of events in the system. Generally this is a small subset of the set of all program variables. The current FLAVERS implementation provides automated support for several different kinds of constraint FSAs. An analyst might need to iteratively add constraints and observe the analysis results several times before determining whether an inconclusive property is truly indicating a fault in the system. Constraints give analysts important control over the analysis process by letting analysts determine what parts of a system need to be modeled more precisely.

2.4 State Propagation Algorithm

FLAVERS uses a fixed-point algorithm that propagates sets of tuples through the nodes of the TFG. Each tuple represents a state of affairs that has been found to be true for at least one path leading to the node through the TFG. The state of affairs summarized by the tuple is a composite of the state that the property FSA will be in and the states that all of the constraint FSAs will be in, when execution of such a path reaches this node [NCO98]. Suppose we wish to verify a property $P = (S_P, \delta_P, s_P^0, A_P, \Sigma_P)$ over a TFG $G = (N, E, n_{\textit{initial}}, n_{\textit{final}}, \Sigma_G, L)$ using a set of constraints $C_1, \ldots, C_k$ 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 $\mathcal{T} = S_P \times S_{C_1} \times \cdots \times S_{C_k}$. A tuple is any $t \in \mathcal{T}$. Define the *initial tuple* as the tuple $T^0 = (s_P^0, s_{C_1}^0, \ldots, s_{C_k}^0)$. Define a transition function $\Delta : \mathcal{T} \times N \rightarrow \mathcal{T}$ as follows

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

where

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

This transition function takes a tuple and a TFG node and produces a new tuple by determining the effect that the event annotating this 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 Δ 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 [MR90].

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. As we are generally concerned only with terminating program executions, only the tuples on $n_{\textit{final}}$ 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 for which every constraint FSA is in an accepting state, ensuring that all feasibility constraints are satisfied. More formally, a violating tuple is $t = (s_P, s_{C_1}, \ldots, s_{C_k})$ 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.

To improve efficiency, when propagating tuples, if a tuple t returned by Δ 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 Δ to t will have C_i in its constraint violation state. Thus,

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

when we examine the tuples on n_{final}, we will discard t' since it corresponds to an infeasible path. Consequently, a tuple with a constraint in its constraint violation state is discarded as soon as it is created, thereby assuring that property FSA states resulting from constraint violations are not propagated forward. Let $\mathcal{T}_V$ be the set of all such tuples with constraint violations.

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

Using the formal definitions, we can provide a state propagation algorithm to verify a property P over a TFG G with constraints $C_1, \ldots, C_k$. 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 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 Δ to m, removing any tuples that have a constraint in a constraint violation state (line 5)[1]. 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.

If a violation is found, FLAVERS can create traces through the model that cause a violation of the property. To create such 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 $n_1, n_2, \ldots, n_l$, such that $n_1 = n_{initial}$, $n_l = n_{final}$ and there exist tuples $t_1, t_2, \ldots, t_l$ such that $t_1 = T^0$, t_l is a violating tuple, and $\forall 1 < i \leq l$: $t_i = \Delta(t_{i-1}, n_i)$. A thorough treatment of the problem of generating such paths can be found in [CCO01].

The FLAVERS state propagation algorithm has worst-case complexity that is $\mathcal{O}\left(N^2 \cdot |S|\right)$, where N is the number

[1]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.

```
class Fork {
  boolean isUp = false;

  synchronized void up() {
    boolean success = false;
    while(! success ) {
      if(isUp) {
        this.wait();
      } else {
        isUp = true;
        success = true;
      }
    }
  }

  synchronized void down() {
    isUp = false;
    this.notifyAll();
  }
}
```

```
class Philosopher extends Thread {
  Fork left, right;

  Philosopher(Fork l, Fork r) {
    left = l;  right = r;
  }

  void run() {
    while(! done) {
      left.up();
      right.up();
      startEating();
      stopEating();
      left.down();
      right.down();
    }
  }
}

class Main {
  public static void main() {
    Fork fork1 = new Fork();
    Fork fork2 = new Fork();
    Philosopher phil1
       = new Philosopher(fork1, fork2);
    Philosopher phil2
       = new Philosopher(fork1, fork2);
    phil1.start();
    phil2.start();
  }
}
```

Figure 2: Dining Philosopher Example

of nodes in the TFG, and $|S|$ is the product of the number of states in the property automaton and the number of states in each of the constraint FSAs. Later in this paper we present some preliminary results that seem to suggest that FLAVERS can verify a large class of important properties using only a small set of constraints. Indeed, these 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, we believe that the FLAVERS analysis technology has the potential to scale to handle real-world sized software systems.

3 Examples

3.1 Example Without Concurrency

We now present an example that shows how FLAVERS can be used to verify properties of a software system and to identify paths on which the property can be violated. The example we present is based upon the Dining Philosopher Problem, which is frequently cited in software analysis literature. The Dining Philosopher Problem details a scenario

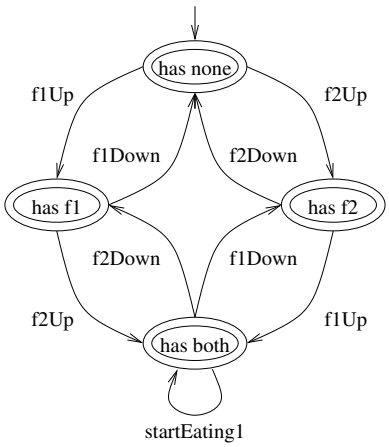

Figure 3: Property `has-two-forks-to-eat`

in which an equal number of philosophers and forks are interleaved alternately around a circular table. Thus each philosopher has exactly one fork on the philosopher's left and one on the philosopher's right. It is further hypothesized that a philosopher can eat only after having picked up both the fork on the left and the fork on the right. Thus, clearly, when one philosopher is eating, both the philosoper to the left and the philosopher to the right will be unable to eat until the dining philosopher finishes eating and relinquishes the forks. Figure 2 is a program that models this situation for the specific case of two philosophers and two forks[2]. Generalization to any number of forks and philosophers is straightforward.

Note that the class Philosopher is defined to be a thread whose execution consists of executing a sequence of two methods (up and down), on two different instances (referred to as left and right) of the class Fork. Specifically, the philosopher invokes the up method first on the left instance of Fork, then on the right instance of Fork. Having done so, the philosopher can then eat (represented by the startEating statement). The philosopher finishes eating, and then invokes the down method, first on the left instance of Fork and then on the right instance of Fork. Presumably, after the first two method invocations, the philosopher is able to eat, and having done so then invokes the next two methods, relinquishing both forks to enable others to eat. This sequence of invocations is nested inside a loop enabling the philosopher to attempt to eat again at a future time.

The class Fork implements the two methods, up and down, whose purpose is to maintain the status of Fork instances, as represented by the boolean variable, isUp. The method Fork.up first checks isUp to see if this instance of Fork is already raised. If not, the method sets isUp to true. If the instance of Fork is already raised (isUp is true), then the method waits for a notification that the instance of Fork has been put down, indicated by the resetting of isUp to false. Indeed, the method Fork.down consists of resetting isUp to false and sending a notification that this has taken place (using the notifyAll method).

Finally, note that the Main class sets the Dining Philosophers Problem in motion by invoking the start method first on phil1, one instance of class Philosopher, and then on phil2, a second instance of Philosopher.

We are interested in demonstrating that this program is indeed a valid model of the behaviors that define the Dining Philosophers Problem. To do so we need to verify that the program must always adhere to key characterizing behavioral properties. Earlier authors have tended to focus on the fact that careless program simulations of the Dining Philosophers Problem can easily lead to runtime deadlocks, and have demonstrated the use of their analyzers in detecting the possibility of such deadlocks. But clearly absence of deadlock is but one of many properties that characterize the Dining Philosophers Problem. We now identify one of these properties and, as an example, demonstrate how FLAVERS can be used to verify that property.

We define the `has-two-forks-to-eat` property to be that a philosopher will not be able to eat until both adjacent forks have been acquired and raised to the up position. While our cursory description of our simulation

[2]In this example, both philosophers pick up fork1 and then fork2, meaning one philosopher picks up the left fork first while the other philosopher picks up the right fork first. This is necessary to prevent deadlock in the system.

program may strongly suggest that our implementation must always satisfy this property, closer inspection reveals that the simulation depends upon the correct use of some flags, variables, and parameter bindings. Thus a careful analysis is indicated.

The goal of our analysis is to show that for all possible executions of this program, at the time each instance of the startEating statement is executed, the corresponding Philosopher class will have both of the instances of the Fork class that were bound as parameters placed into the isUp state. We thus need a property FSA representing the desired up/down status of the two forks. To support the reasoning needed to assure this, we model the acquisition/raised status of a philosopher's adjacent forks by a property FSA shown in Figure 3[3].

The `has-two-forks-to-eat` property FSA has four states, representing the four possible combinations of the states in which the two forks adjacent to the philosopher can be at any time. These states are no forks raised, either of the two forks raised, or both of the forks raised. Note in addition, that transitions between these states are driven by the events of picking up and putting down forks. Thus, specifically, note that the f1up event drives the FSA from the has none state to the has f1 state, from which the subsequent f2up event will drive the FSA to the has both state. Once in the has both state, subsequent events representing the putting down of forks will drive the FSA toward the has none state.

To relate an execution of the program to transitions of the FSA between its various states, it is necessary to annotate the program by indicating which program statements affect the various events that drive the FSA. There is some challenge in doing this because the various program statements represent the specific fork instances upon which they operate symbolically, rather than explicitly. We need to know which specific fork instance is being raised or lowered by each of the various statements in the program. Thus it is necessary to create a representation of the program that separately represents different class instances, and dereferences the uses of all symbolic names relating to Philosopher and Fork class instances. This enables us to represent exactly which methods are being applied to which fork instances at which locations in each instance of the Philosopher class. Thus, during inlining, when we replace each method invocation with the text of the method invoked, we substitute the actual argument for each instance of each of the method's formal parameters and create explicit references to different instances for all variables that are local to the invoked instance.

Figure 4 is an example of a control flow graph (CFG) model after this inlining process has been applied. This particular flowgraph represents the inlining of the method invocation phil1.start() from procedure Main. The ovals correspond roughly to the statements that are executed as a consequence of this method call. Each oval is annotated with three lines. The top line is a unique node identifier. The middle line denotes the thread in which the originating invocation is located (in this case Philosopher1, the first instance of class Philosopher). The bottom line represents the type of action effected by the execution of the statement represented by this node. Thus, note that node 3 represents execution of a while statement. Accordingly note that there are two outedges from this node, representing looping and loop termination. Nodes 5, 6, and 7 represent the execution of different synchronization functions. They model progress from entering the state where Philosopher1 will wait for fork1, to the state where Philosopher1 is waiting for fork1, to the state where Philosopher1 is been notified that fork1 is available and the wait state can be left. These nodes are not particularly important to the analysis currently being described. Their importance will be addressed in the context of a subsequent example. Finally, note that node 8 represents the action of having fork1 raised.

This last node is of particular interest, because it represents the f1Up event that drives the `has-two-forks-to-eat` property FSA from one state to another. The other four events used by the `has-two-forks-to-eat` property FSA take place in statements represented by node 16 (the f2Up event), 18 (the startEating1 event), 23 (the f1Down event), and 28 (the f2Down event). It is particularly noteworthy that relatively few nodes in this CFG carry annotations found in the FSA. Thus, Figure 5 represents the TFG resulting from reducing the CFG in Figure 4 to reflect only events used in the property FSA `has-two-forks-to-eat`.

Having specified the property FSA `has-two-forks-to-eat`, and developed the TFG it is possible to initiate the verification. An initial expectation might be that FLAVERS would verify that it is impossible to reach the startEating statement with the FSA in any state other than the desired has both state. But instead, FLAVERS reports

[3]FLAVERS requires that FSAs be total, meaning they have a transition from every state on every node in the alphabet. The FSA in Figure 3 and the other FSAs we will show are not total, to make them more compact and easier to understand. They can easily be made total, by adding a *trap* state that is non-accepting and has all self loop transitions and by adding transitions from the existing states of the FSAs to this trap state on events where transitions do not already exist.

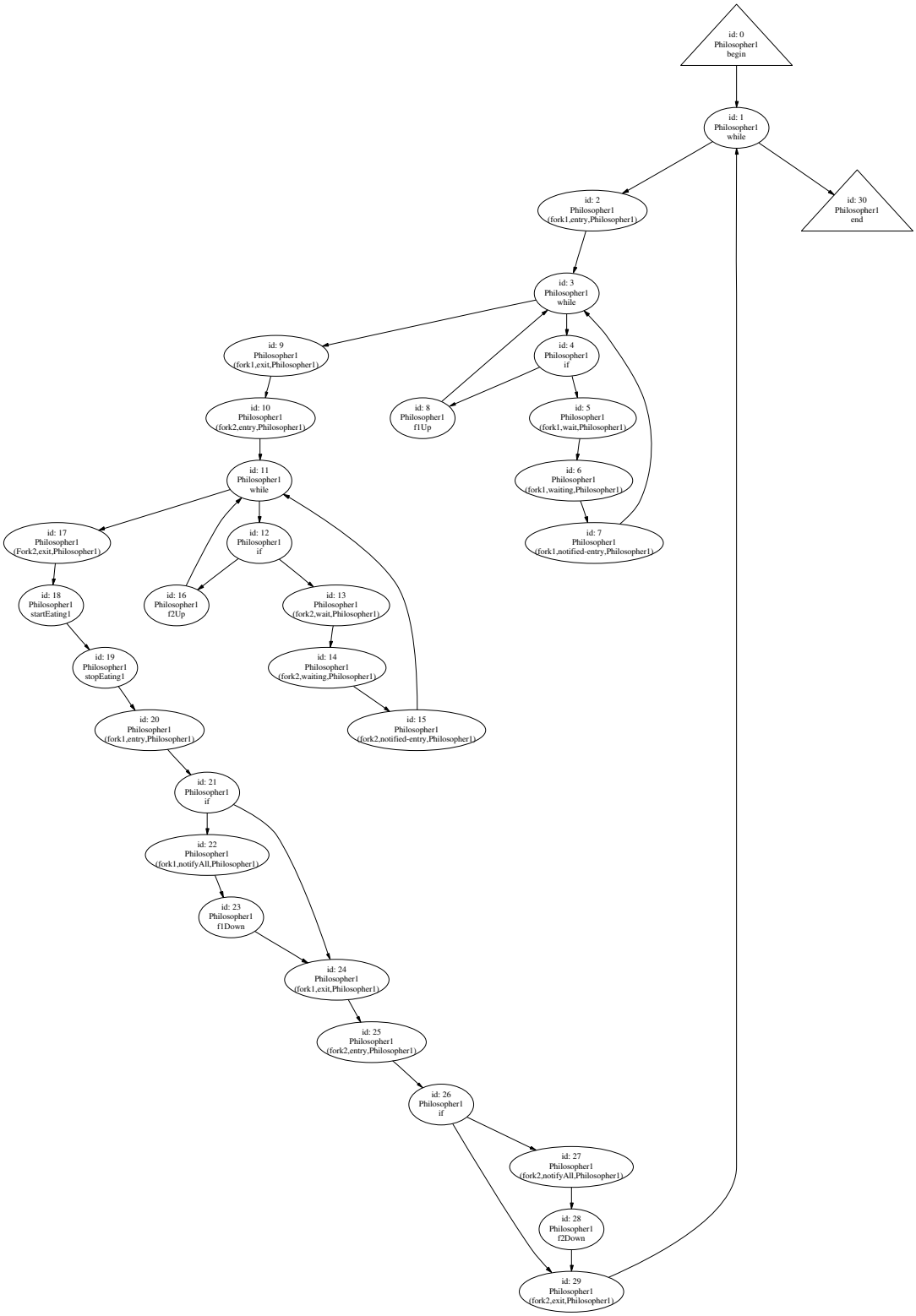

Figure 4: Control Flow Graph for `Philosopher1.run()`

Figure 5: TFG for example

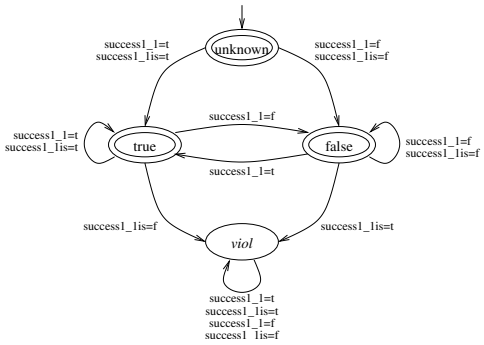

Figure 6: Variable Automata for `success1_1`

inconclusive results, returning the path $0 \rightarrow 1 \rightarrow 18 \rightarrow 23 \rightarrow 28 \rightarrow 30$ as a violating path, namely a path whose execution would leave the FSA in a state other than has both. Looking back at the original CFG in Figure 4, we can see that the only way to get from node 1 to node 18 while avoiding node 8 involves exiting the loop headed by statement 3 (the representation of the statement while(! success1_1)) without setting success1_1 to true. Again looking at the original program text we see that this cannot happen because success1_1 is always set to false in a statement represented by node 2, and it not reset to true anywhere except in a statement represented by node 8. But it is easy to see how FLAVERS will not recognize this, because the CFG in Figure 4 does not represent those specific program details. As noted above, such details are suppressed in the interests of reducing graph size and accordingly substantially reducing execution time. But, as noted above, these efficiencies can lead to loss of analytic accuracy of just the sort that we have just observed.

But, as was stated earlier, feasibility constraints provide a way to incrementally improve accuracy by supporting the removal of infeasible paths from consideration. This is done by introducing constraint FSAs, whose consideration increases graph size and can increase execution time, but also increases analytic precision. In this case, we want to model the value of the variable success1_1 by the constraint FSA shown in Figure 6.

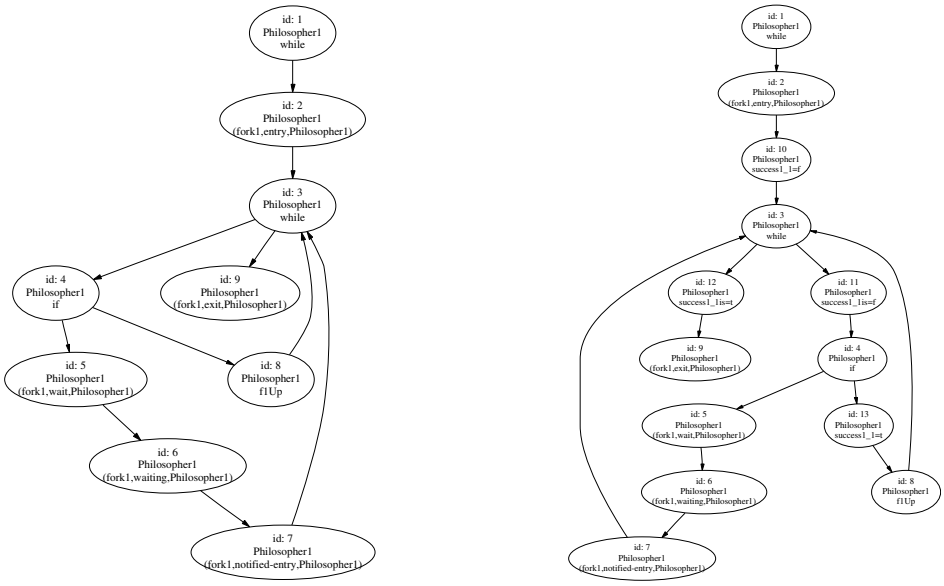

Figure 7: Adding variable events to CFG

The event alphabet of this automaton has four events, the two representing the two possible assignments of a value (true or false) to the variable, success1_1=t and success1_1=f, and two representing tests of the value of the variable that return the two possible different values, success1_1is=t and success1_1is=f. This automaton has four states, one of them represents the state of being certain that the variable's value is definitely true. One of them represents the state of being certain that the variable's value is definitely false. One of them represents the situation where the analysis cannot be certain of the variable's value. And one represents the constraint violation state, which is entered when an event sequence is infeasible with respect to this variable. Thus, execution of a statement assigning a literal value to the variable transitions this constraint automaton to one of the two states where the variable's value is known. But, for example, when the constraint FSA is in the state true and the event success1_1is=f is seen, then this means a reference is being made where the referenced value is assumed to be false. This inconsistent behavior is impossible in any actual program execution, and this is represented by the transition of the constraint automaton to the constraint violation state. As noted above, when the FLAVERS state propagation algorithm determines that a constraint automaton has transitioned into a violation state it uses this information to decline to consider event sequences along further extensions of this path. For FLAVERS to use this approach to sharpen the accuracy of its analysis of the program in Figure 2, it is necessary to build a TFG containing nodes that represent all statements at which all events in the alphabet of the constraint FSA depicted in Figure 6 occur.

Figure 7 shows a fragment of the CFG from Figure 4 and the same fragment after it has been augmented with nodes included specifically to represent statements at which events from this additional alphabet occur. When the property is checked using this constraint FSA and this augmented TFG, FLAVERS again returns an inconclusive result. As before, this is due to infeasible paths, this time because the variable success1_2 was not modeled. Once a constraint FSA representing this second variable has been created, and once a TFG incorporating the additional events in the alphabet of this automaton has been created, FLAVERS can finally return a conclusive result for the property `has-two-forks-to-eat`.

It is important to note that FLAVERS is able to automatically generate some constraint FSAs for modeling the values of variables and some augmentation to the TFGs to include representations of the events in the alphabets of such automata. Although these more complex and precise analyses may increase the cost in terms of graph size and execution time, sometimes they reduce the search space so as to reduce the overall analysis cost.

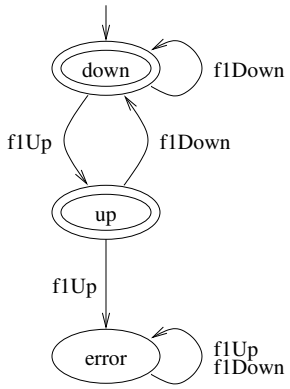

Figure 8: Property `no-fork-raised-twice`

3.2 Example Requiring Analysis of Concurrency

Our first example property could be studied by analyzing the code for only one thread. There will be some properties of this sort in a concurrent program, but quite often it will be the case that some important properties can be studied only by taking into account the combined effects of different threads of control.

As an example we now consider the property `no-fork-raised-twice`, namely that no fork could ever possibly be raised more than once. While we could prove this property over all forks in the system simultaneously, it is sufficient and easier to prove this property for an arbitrary fork. Since all the forks are treated identically, an argument can be made that proving this property for any one fork is sufficient. The FSA for the property is shown in Figure 8 and has been made specific to fork1. This FSA represents the property by showing that if the event f1Up occurs twice in succession, without an intervening f1Down event, then the automaton moves into an error state.

In the previous example we then developed a TFG representation of the program by inlining the procedures invoked through methods that might be capable of causing a violation of the property. For this new property, however, such a straightforward approach will not work. The violation of interest, raising a fork more than once, might occur if a given fork were raised from each of two different threads of control. Thus inlining the invocations of fork methods in only one thread of control (as was done in the previous example) will not represent the full range of behaviors that might potentially cause the violation of the property.

Indeed, properties such as this require the representation of all possible interleavings of events taking place on all threads that might execute in parallel. We represent all possible interleavings by constructing a TFG in which all parallel threads and tasks are represented as separate CFGs that are then interconnected by MIP edges. We illustrate this by developing the TFG needed to verify this property. Figure 9 shows the start of this TFG. It has one subgraph for each thread that can execute in parallel in our program, namely a Main object thread and one thread for each of the two philosophers. To get these subgraphs, we removed nodes from the CFG that did not contain events related to the property `no-fork-raised-twice` or did not affect the flow of control over these nodes. Nodes representing concurrency control affect the flow of control and are now of critical importance as they are used as the basis for determining the different ways in which interleavings of events across concurrent threads can occur.

In particular, we use such nodes to identify sets of nodes from one CFG that might execute in parallel with sets of nodes from another CFG. Thus, for example, note that nodes 1 and 23 can execute in parallel, but nodes 6 and 28 cannot. Indeed, assuming that node 6 has executed, but that node 17 has not yet executed, it is necessary that the execution of node 24 will be followed by executing nodes 25 and 26. Indeed the execution sequence 24, 25, 26 may happen in parallel with the execution sequences that begin with 6 and end with 17. Conversely, if statement 28 executes before statement 6, then statement sequence 3, 4 will immediately follow 2, and may happen in parallel with sequences beginning with 28 and ending with 39. There are numerous other such pairs of execution sequences that may happen in parallel among these three threads. In general the computation of precisely which statement sequences may happen in parallel with which others is quite complex. FLAVERS incorporates an algorithm [NA98, NAC99b] that computes these pairs automatically based upon CFGs annotated very much as shown in Figure 9. Once the sequences

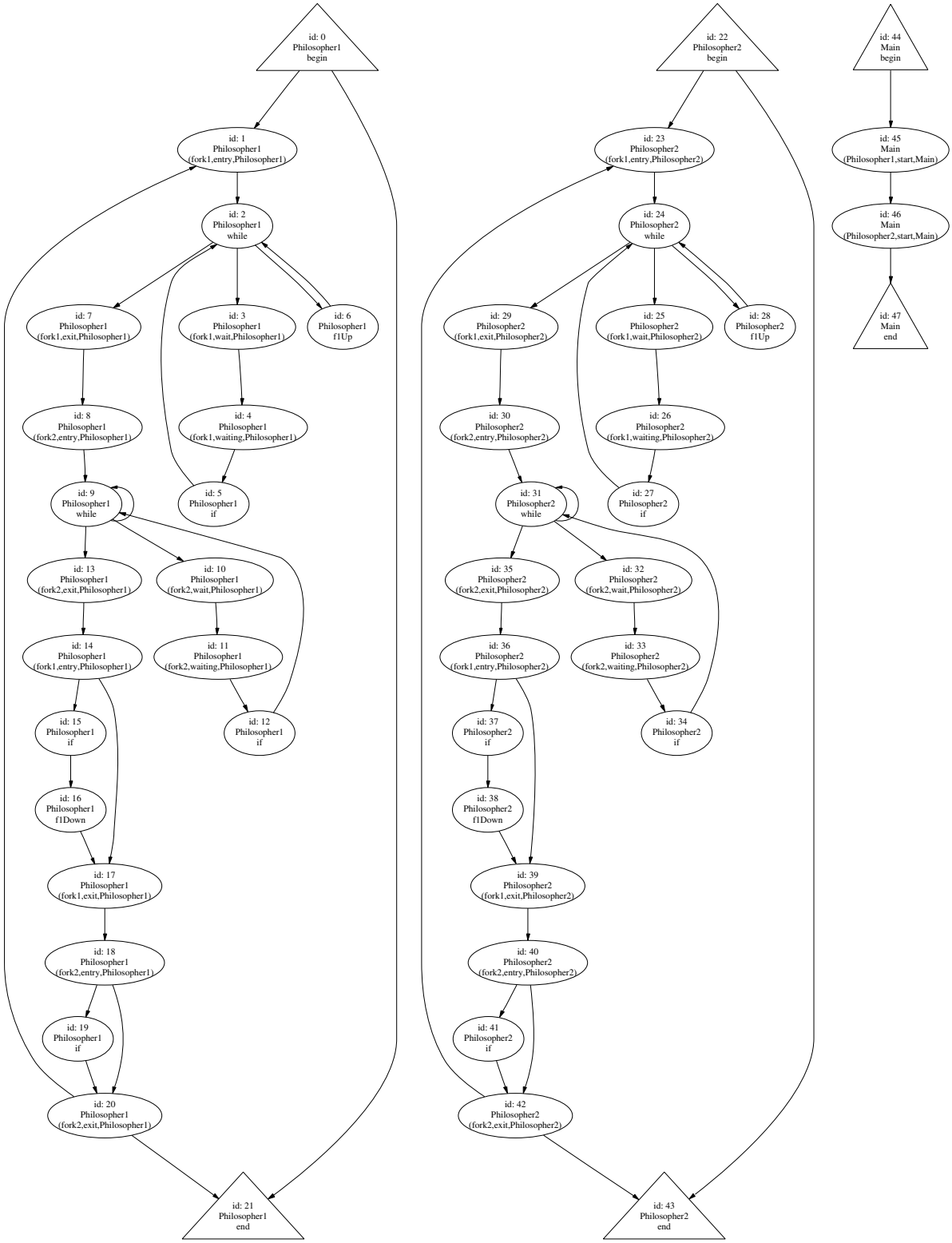

Figure 9: Start of TFG for `no-fork-raised-twice`

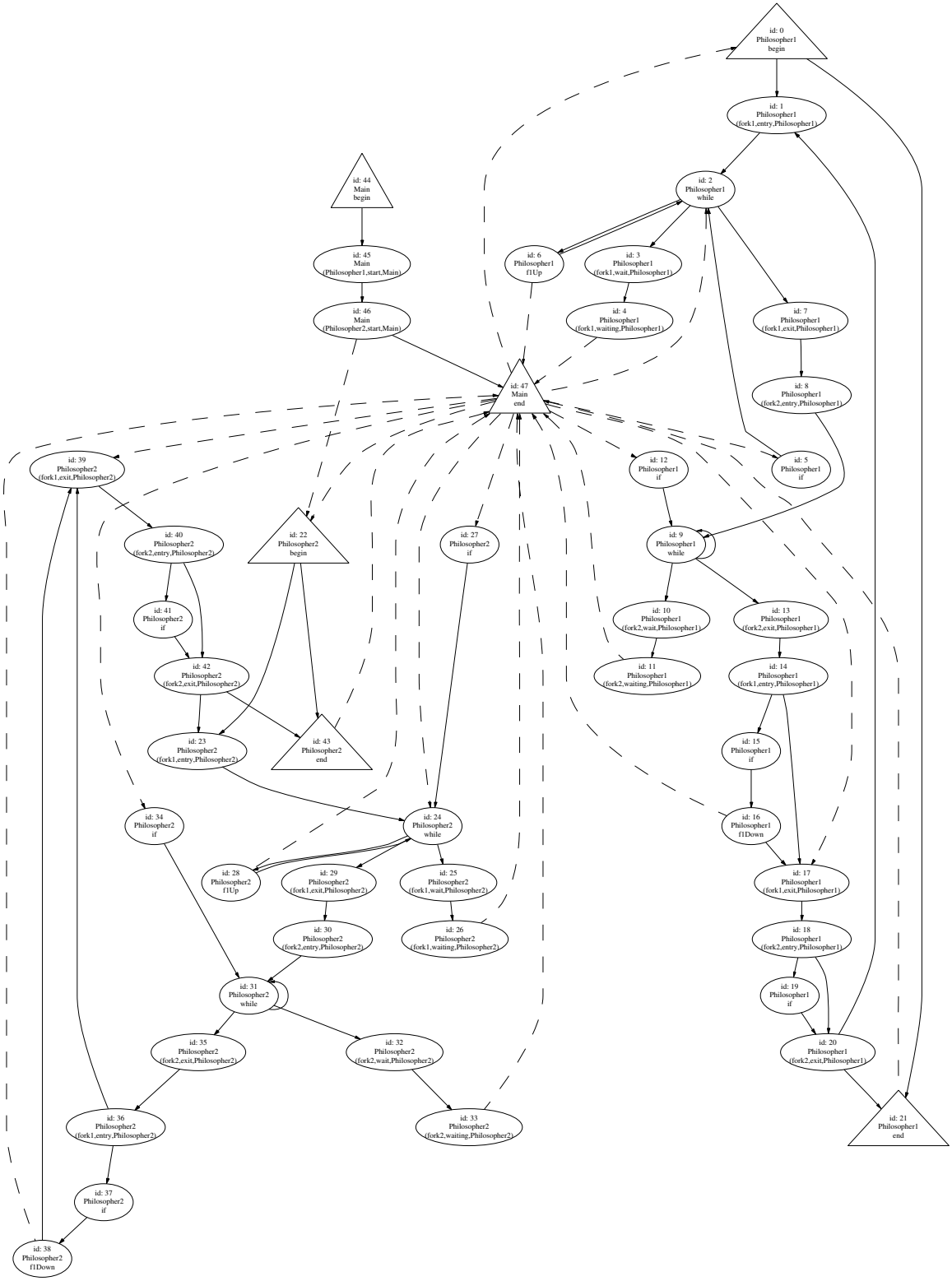

Figure 10: TFG for `no-fork-raised-twice` with MIP edges for node 47

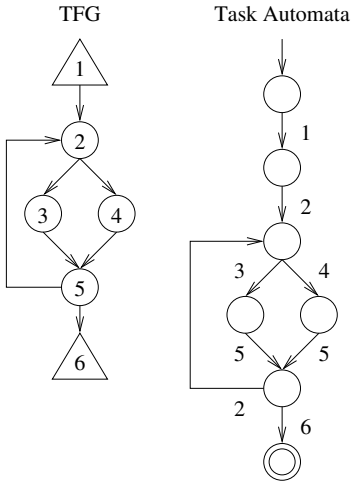

Figure 11: Task Automata Example

of statements that may happen in parallel have been computed, it is rather straightforward to add MIP edges to form the required TFG. Every node in one statement sequence must be connected to every node in every other statement sequence with which the initial statement sequence can happen in parallel[4]. For all but the most trivial concurrent programs, the number of MIP edges can be expected to be very large. As an example, in Figure 10 we have used dashed lines to show only the set of MIP edges that are incident with node 47.

Having developed the graph structure containing all possible MIP edges and with the property automaton representing the property `no-fork-raised-twice`, it is now possible for FLAVERS to determine whether a given fork can be raised twice. Execution of FLAVERS using this TFG fails to verify the property, and instead returns a counter example path $44 \rightarrow 45 \Rightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 6 \rightarrow 2 \rightarrow 6 \Rightarrow 47$. In this path, nodes connected by $\rightarrow$ represent the consecutive execution of two statements that are in the same thread of control. Nodes connected by $\Rightarrow$ represent the consecutive execution of two statements that are in different threads of control, but which are connected by a MIP edge, indicating that this interleaving is possible. Examination of this counter example path reveals that it infeasible because it goes from node 6 to node 2, but then immediately back to node 6 again. This statement sequence is unexecutable because, as noted in the previous example, the variables success1_1 and isUp1 are used to explicitly prevent this behavior. Thus, as in the previous example, we use a variable automaton, in this case one that models the variable isUp1, to eliminate paths that are rendered unexecutable by the infeasible usage of this variable.

Having incorporated this variable automaton into the analysis, FLAVERS still returns an inconclusive result, in this case, producing a counter example that includes a subpath that goes from node 6, the node in Philosopher 1 with event f1Up, to a node in another thread via a MIP edge, and then immediately back again, forcing the property to the error state. This counterexample highlights a serious problem in the unrestricted use of MIP edges in our analyses. The problem is that an arbitrary path in a TFG including MIP edges may not be executable, just as an arbitrary path in a CFG may not be executable. For example, after node 6 has executed in the Philosopher1 thread, the next node in Philosopher1 to execute must be node 2. It is certainly impossible for node 6 to execute again. Unfortunately, jumping from node 6 to a node in a parallel thread takes us to a node that is connected back to node 6 by a MIP edge. To suppress the consideration of these kinds of illegal path sequences, FLAVERS employs the use of yet another type of feasibility constraint, referred to as a task automaton. The purpose of a task automaton is to assure that every path considered in the analysis through the various threads, is a reasonable path within that single thread. As an example, consider the TFG fragment and the corresponding task automaton shown in Figure 11. The alphabet of the task automaton is the set of ID numbers for the nodes in the TFG. The task automaton has one more state than the TFG has nodes and ensures that the first of this task's nodes that is visited in any execution sequence is node 1, and that the next of this

[4]We can do better than this by using a partial order optimizaion, which can reduce the number of MIP edges by noting that certain interleavings are equivalent with respect to the property and constraints. Thus, it is only necessary to consider one interleaving from each equivalence class [NCC99].

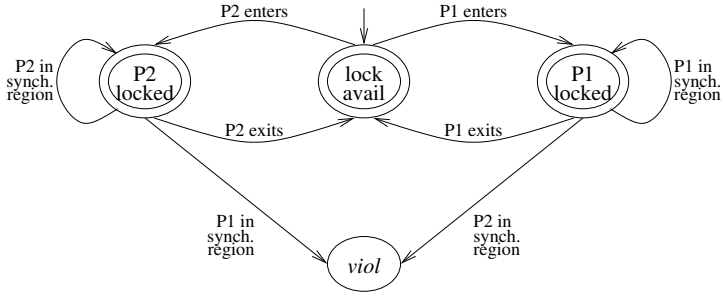

Figure 12: Monitor Feasibility Constraint

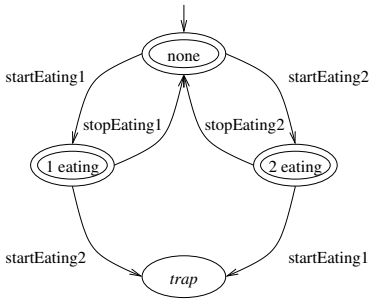

Figure 13: Property `no-eat-at-same-time`

task's nodes in any execution sequence has to be node 2, and so on. The task automaton also ensures that no execution sequence will specify execution of any node in this thread subsequent to the execution of node 6, by incorporating a transition on 6 to the only accepting state in the FSA.

When we perform the analysis with the task automaton for the Philosopher1 thread, we get inconclusive results. The counter example FLAVERS returns is similar to the previous one, except the tasks are reversed. To correct this, we add a task automaton for the thread Philosopher2. Executing the analysis with these task automata still yields inconclusive results, but the counterexample clearly indicates the need for one final constraint automaton to model the synchronized regions on fork1.

When we introduce a feasibility constraint to ensure the lock is respected, (the template for this constraint is shown in Figure 12), FLAVERS returns conclusive results for the property `no-fork-raised-twice`.

These two properties are representative examples of the sorts of properties that can and should be verified to show that the program is a valid implementation of the Dining Philosophers problem. Many others can readily be suggested. For example, two adjacent philosophers cannot both be eating at the same time. Again, this property could be proved over all philosophers in the system, but it is easier to make it specific to a pair of philosophers. The FSA for this property is shown in Figure 13. It has three states to keep track of who is eating at any given time. If it is in a state where one philosopher is eating and another philosopher starts to eat, then it is driven into the property violation state. In order to prove this property conclusively, we need variable automata for isUp2, success1_2, success2_2, and task automata for Philosophers 1 and 2 and a monitor constraint for fork2. These six constraints are sufficient to prove this property for this system with any number of philosophers.

Note that task automata and concurrency control FSAs can be easily automatically derived from the CFG. Right now, the analyst must indicate which of these constraint automata to include when doing a verification problem, however, we expect that heuristics could be developed to make reasonable choices automatically.

Some other example properties are: a fork cannot be put down unless it is already up, a fork that is down cannot be put down, a philosopher must put down both forks after completing the startEating step. In each of these cases we can readily define a property automaton to capture the property, can demonstrate a TFG to support verification of the property, and can specify constraint automata sufficient to cause the analysis to produce definitive results.

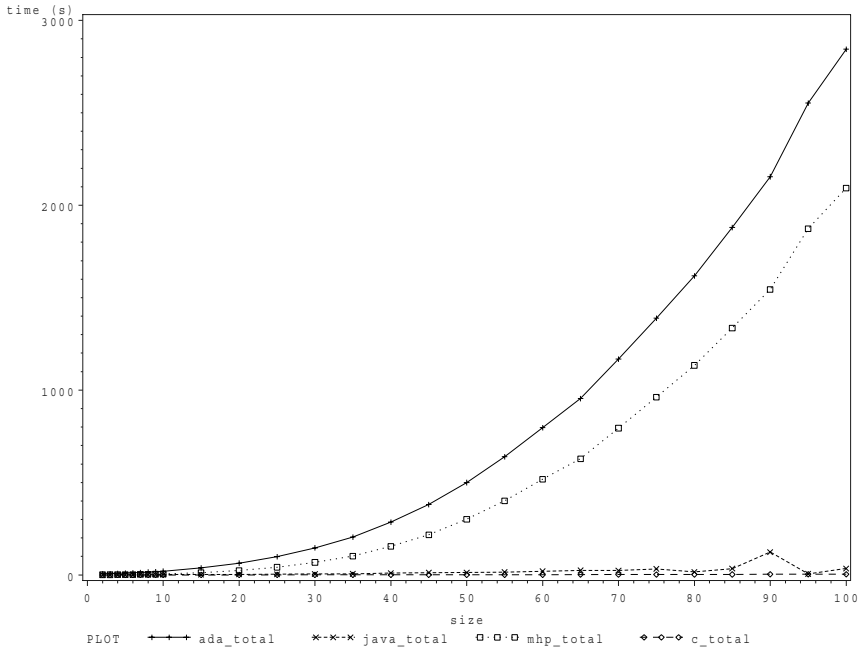

Figure 14: Timing for `no-fork-raised-twice`

4 Some Experimental Results

Our FLAVERS prototype for analyzing programs written in Ada is considerably more mature than the prototype for Java. Therefore all the results presented in this section are based on FLAVERS/Ada. The Ada version of our example problem for n philosophers has $2n$ tasks, one task for each philosopher and one task for each fork. In this section we present the timing results for analyzing the dining philosopher problem for the two properties `no-fork-raised-twice` and `no-eat-at-same-time`. We show how FLAVERS performs as we increase the number of philosophers, and consequently the number of forks.

To prove the property `no-fork-raised-twice` conclusively for the Ada version of the program only required a task automata for the fork1 task. To prove the property `no-eat-at-same-time` conclusively for the Ada version of the program required three constraint automata, one for philosopher 1, one for philosopher 2, and one for fork 2, their shared fork. For both these problems, as we increased the number of philosophers and forks, we did not need to increase the number of feasibility constraints needed to prove this property conclusively. We have found that this is often the case. Once the necessary feasibility constraints are found for a small configuration of a system, it is often the case that the system can be scaled without having to add additional feasibility constraints.

To prove properties of actual source code, it is necessary to first use language processing tools to translate the source code into annotated CFGs. For FLAVERS/Ada the translator is written in Ada and built on the Arcadia infrastructure components described in [TBC+88]. Tools written in Java are then used to translate these CFGs into a TFG and to construct finite state automata representations of the properties and any feasibility constraints. Once all of this has been done, the FLAVERS state propagation algorithm is used to verify the property. To maximize execution speed, the state propagation algorithm is written in C.

The time measurements given here are sums of the user and system times as measured by `/usr/bin/time` on a Sun Enterprise 3500 with two 366 MHz processors and 2 GB of memory running Solaris 2.6. While this is a multi-user system, for all experiments, we had exclusive access to the machine to prevent variance in the times caused by other users on the machine. The Ada portion of the FLAVERS/Ada tools were compiled using the Verdix Ada Development

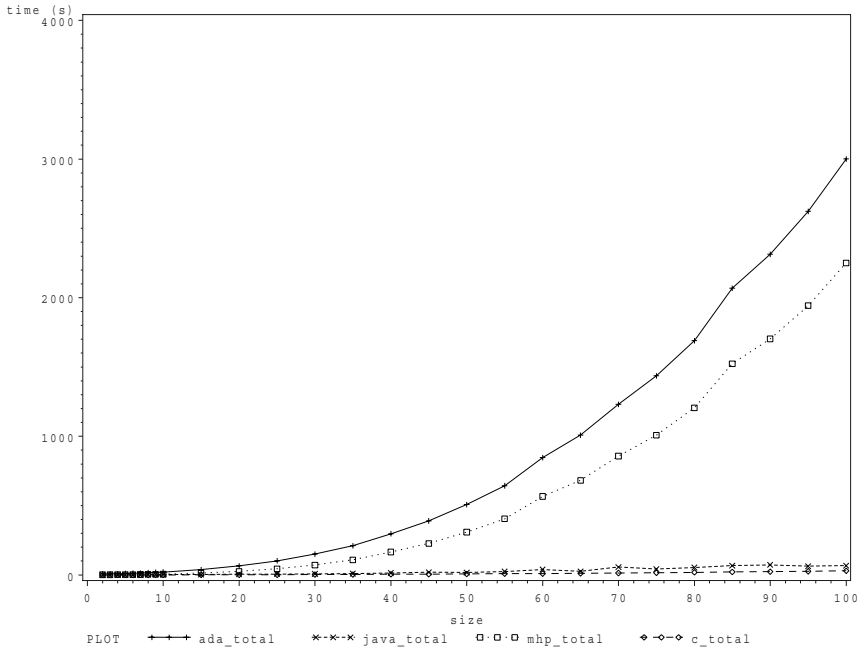

Figure 15: Timing for `no-eat-at-same-time`

System version 6.2.3c with optimizations disabled (to avoid known compiler bugs). The Java portion of the tools was run using the Sun JDK version 1.1.8. The C version of state propagation was compiled with the Free Software Foundation's gcc version 2.95.2, using -O2 for optimization.

The timing results for these properties are shown in Figures 14 and 15. The x-axis shows the number of philosophers and the y-axis shows the running time in seconds. Each of these figures has four lines. The c_total is the total time for running state propagation. The java_total includes this time, plus the time for running all of the Java tools except for the MHP analysis. The mhp_total includes these times, plus the time for running the MHP analysis. The ada_total includes these three times, plus the time for running the Ada language processing tools, making this the total time of the analysis, from start to finish.

As can be seen from these two figures, state propagation is surprisingly efficient. The Ada processing is expensive and probably unnecessarily so. The Ada front-end is built upon a front end that is now obsolete. A re-implementation should be expected to improve these speeds considerably. The time to compute the MIP edges is the primary expense. For Ada this algorithm has worst-case complexity that is $\mathcal{O}\left(S^6\right)$, where S is the number of statements in the program. For a problem like dining philosophers, where there is much interaction between the tasks, we might be seeing a performance that is close to the worst-case bound. It is interesting to note that the algorithm for computing MIP edges for Java has worst-case complexity that is $\mathcal{O}\left(S^3\right)$, so we would expect the overall performance to be somewhat better. Even with this prototype version of the system, the overall performance was less than 50 minutes for 100 philosophers and the actual state propagation time was less than 1 minute for 100 philosophers in the worst case.

In an attempt to estimate the actual functional dependence between running time and the number of philosophers, we fit different polynomials to the ada_total lines shown in Figures 14 and 15. The results of these fittings are shown in Table 1. Each column gives the data for the best-fit polynomial of the given form. In each column, r^2 is the percentage of variance in the data explained by the polynomial. The remaining rows give the coeffiecients of the best-fit polynomial.

For each problem, the more terms in the polynomial, the better the fit as measured by r^2. This is not unexpected, because each additional term adds a degree of freedom to the fit. In both cases, a linear polynomial explains the data

no-fork-raised-twice			
	$c_1x + c_0$	$c_2x^2 + c_1x + c_0$	$c_3x^3 + c_2x^2 + c_1x + c_0$
r^2	.8778	.9986	0.9998
c_0	-333.8924	66.9270	13.4860
c_1	24.6496	-9.8018	-1.2734
c_2		0.3707	0.1451
c_3			0.0015

no-eat-at-same-time			
	$c_1x + c_0$	$c_2x^2 + c_1x + c_0$	$c_3x^3 + c_2x^2 + c_1x + c_0$
r^2	0.8760	0.9987	0.9996
c_0	-354.6604	71.7946	21.6711
c_1	26.0000	-10.6548	-2.6564
c_2		0.3944	0.1828
c_3			0.0014

Table 1: Polynomial fitting

well, but adding a quadratic term improves the fit considerably. Adding a cubic term improves the fit, but not by much. As a result, it appears that this data is best explained by a quadratic polynomial.

Admittedly, the dining philosopher program is a small and contrived example. It tends to have more task interaction than most concurrent programs, so it is not an unreasonable example to study. It also is easy to understand and easy to scale. The profiles of the timing diagrams for the two properties examined here are, however, representative of the timing diagrams that we tend to see when the current FLAVERS prototype is applied to a variety of other systems. Although the prototype could be improved considerably, the performance indicates that the approach seems to scale well.

In [ACD+99], a comparison was made of several finite state verification tools on a real software system, called Chiron [FMCB93]. In that study, SPIN [Hol97], SMV [McM93], INCA [ABC+91] and FLAVERS were all applied to the same problems. Although there was considerable variation in how the tools performed from problem to problem, the computation time associated with FLAVERS, with a few exceptions, grew at a lower rate than the other tools [5].

5 Related Work

Flow analysis techniques for verification were originally used to detect potential definition/reference anomalies in sequential code [OF76]. This approach was later extended to allow the verification of user-specified properties [OO90, OO92]. FLAVERS extended this work to support verification of concurrent systems and to support incremental improvements to the model of the system being analyzed [DC94, DC99].

Most other approaches to finite state verification have been based on building a reachability graph model of the software system and thus have a worst-case bound that is exponential in the number of tasks in the system. SPIN [Hol97] analyzes systems written in Promela and creates a highly optimized representation. SMV [McM93] and other model checkers (e.g., [God97]) use abstractions, such as binary decision diagrams, and optimization, such as partial order reductions, to reduce the size of the system model. These optimizations and abstractions have effectively reduced the model size for many impressive examples.

INCA [CA95] is a FSV approach that is not based on a reachability graph model. INCA creates inequalities that describe the legal flow through the system and models the complement of the property as an inequality as well. It then uses integer linear programming techniques to determine if there exists a solution representing flow through the system that is a violation of the property. Although, in general, integer linear programming has an exponential worst-

[5] In this experiment, FLAVERS often had the worst performance on the smaller size problems but it was the only FSV system that was analyzing source code instead of a model of the source code.

case bound, the inequalities generated by INCA are usually extremely simple so that the performance is quite good. INCA assumes that the system being analyzed uses a synchronous model of concurrent execution; it is currently not clear how to extend this approach to asynchronous execution models.

FLAVERS uses a graph model of the software system that is considerably less precise than the reachability graph approaches described above. Reachability graph models tend to enumerate all the states that a system can be in, where a state indicates the program counter for each task and the particular values of each variable. The optimization techniques employed by these approaches remove information whenever they can, but most models are still precise (and thus conservative) with respect to the property that is being verified. In contrast, FLAVERS uses a model that is conservative but not precise with respect to the property. If it can be shown that the property is consistent with this model, then it is indeed valid. If inconclusive results are returned, then the analyst needs to determine if the system has a fault or if additional information should be added to the model before the analysis is rerun. If an analyst ends up fully modeling all the variables in the system, then the two approaches are basically equivalent. Reachability approaches, however, create models that are often too large to be applied to software systems directly. FLAVERS has an advantage in that precision can be added incrementally, guided by previous analysis results. In the future, we hope to improve and automate more of this guidance. Based on our scaling experiments, it appears that FLAVERS can support actual software systems.

6 Conclusions

Software systems are an integral part of our basic societal infrastructure, playing a role in vital applications such as communication, transportation, finance and medical informatics. With their wide-scale use across the Internet, systems must meet more stringent requirements and perform more reliably than ever. Although testing provides valuable assurances about how a system performs, it cannot guarantee that a system will always adhere to important behavioral requirements. FSV approaches can be used to make such assurances, yet are not as difficult to use as more traditional theorem-proving based formal verification techniques.

Most FSV verification approaches base their analysis on a very precise model of the system and have thus focussed on analyzing hardware or system designs, where the resulting model tends not to be too large. They employ sophisticated optimization techniques to reduce the model, but even then must often rely on users to help find suitable model abstractions to help keep performance tractable.

FLAVERS, on the other hand, has been designed to verify software systems. It relies on relatively standard compiler front-end representation and optimization techniques to create a control flow graph based model of the system. Concurrency is represented using MIP edges to represent potential interleaved execution, instead of enumerating all possible system states. This model basically trades off compactness for precision. If subsequent analysis demonstrates that more precision is required to achieve conclusive results, the analyst directs the system, via feasibility constraints, to add more detail. Thus, the model is built up incrementally, with some support for the automatic creation of commonly used types of feasibility constraints. Our experimental results indicate that this approach is effective at verifying a wide range of event based properties, (although it is not effective at detecting deadlock, which requires a more precise execution model).

The underlying models that FLAVERS employs, an FSA for representing properties and an annotated control flow graph for representing systems, are relatively general. Thus FLAVERS had been applied to different programming languages and property specification languages. For example, in addition to our FLAVERS/Ada and FLAVERS/Java prototypes, others have developed systems for Jovial and C++.

Our two prototypes have clearly demonstrated proof of concept. FLAVERS/Ada is the more mature of the prototypes and has been used to verify small to medium sized systems. Unfortunately, the cost of such verification seems to be very dependent on the actual system and properties being considered [CCA96]. Systems that have events on most statements and have a great deal of inter-task communication, tend to require considerably more time and space for the analysis. Small changes in a program or in a property can have a large impact on the resource utilization, however, making it impractical to predict the cost for a particular analysis.

Observing how a technology handles systems that can be arbitrarily increased in size is one indication of how a system will scale. In this regard, FLAVERS seems to have a growth rate that is usually sub-cubic in the number of

tasks. This compares very favorably with other finite state verification approaches that we have studied.

There are several areas of future research that we intend to explore. We are very interested in further developing the FLAVERS/Java prototype. Java provides a number of interesting language constructs that need to be investigated further. Although we have developed an approach for dealing with the eclectic set of concurrency constructs in Java [NAC99a], more experimentation needs to be done.

There is some interesting work on automating many of the model abstractions used by FSV techniques [CDH+00]. These new abstractions appear to enable other FSV approaches to be more effective in scaling up to sizes needed in order to be effective in analyzing software systems. We expect to evaluate these abstractions to see if they also benefit the system model that FLAVERS uses.

We are investigating ways in which to make specifying properties easier. Although we have tried to support notations that are more natural for practitioners to use than predicate calculus or temporal logic [Pnu77, Eme90] based notations, it is still difficult to capture a property specification precisely. Building upon the work in specification patterns [DAC99], we are trying to develop natural language templates that help analysts understand and select among the choices associated with each pattern.

Finally, we are exploring compositional approaches to the analysis. Although our current techniques for optimizing the TFG have been surprisingly successful, they clearly will not work for systems of any arbitrary size. It is clear that we need to find ways in which subsystems can be verified and the results then combined.

References

[ABB+96] Richard J. Anderson, Paul Beame, Steve Burns, William Chan, Francesmary Modugno, David Notkin, and Jon Damon Reese. Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7):498–520, July 1996.

[ABC+91] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[ACD+99] George S. Avrunin, James C. Corbett, Matthew B. Dwyer, Corina S. Păsăreanu, and Stephen F. Siegel. Comparing finite-state verification techniques for concurrent software. TR 99-69, University of Massachusetts, Department of Computer Science, Nov 1999.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principes, Techniques, and Tools*. Addison-Wesley, 1986.

[BCM+92] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computing*, 98(2):142–170, 1992.

[CA95] James C. Corbett and George S. Avrunin. Using integer programming to verify general safety and liveness properties. *Formal Methods in System Design*, 6(1):97–123, January 1995.

[CCA96] A. T. Chamillard, Lori A. Clarke, and George S. Avrunin. An empirical comparison of static concurrency analysis techniques. TR 96-84, University of Massachusetts, Department of Computer Science, May 1996.

[CCO00] Jamieson M. Cobleigh, Lori A. Clarke, and Leon J. Osterweil. Verifying properties of process definitions. In *Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 96–101, August 2000.

[CCO01] Jamieson M. Coblegh, Lori A. Clarke, and Leon J. Osterweil. The right algorithm at the right time: Comparing data flow analysis algorithms for finite state verification. In *Proceedings of the* 23^{rd} *International Conference on Software Engineering*, May 2001.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Păsăreanu, Robby, and Hongjun Zheng. Bandera: Extracting finite-state models from Java source code. In *Proceedings of the* 22^{nd} *International Conference on Software Engineering*, June 2000.

[CW96] Edmund M. Clarke and Jeanette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[DAC99] Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 16–22, May 1999.

[DC94] Matthew B. Dwyer and Lori A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, December 1994.

[DC99] Matthew B. Dwyer and Lori A. Clarke. Flow analysis for verifying specifications of concurrent and distributed software. TR 99-52, University of Massachusetts, Department of Computer Science, August 1999.

[Dij] E. W. Dijkstra. *Notes on Structured Programming*, chapter 1, pages 1–82.

[Eme90] E. Allen Emerson. Temporal and modal logic. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 995–1072. Elsevier Science Publishers, 1990.

[Flo67] R. Floyd. Assigning meaning to programs. In *Symposium on Applied Mathematics*, pages 19–32, 1967.

[FMCB93] K. Forester, C. MacFarlane, M. Cameron, and G. Bolcer. Chiron-1 user manual. Arcadia Document UCI-93-07, University of California, Irvine, September 1993.

[God97] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proceedings of the* 24^{th} *ACM Symposium on Principles of Programming Languages*, pages 174–186, January 1997.

[Hol97] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[McM93] K. L. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, 1993.

[MR90] Thomas J. Marlowe and Barbara G. Ryder. Properties of data flow frameworks: A unified model. *Acta Infomatica*, 28:121–163, 1990.

[NA98] Gleb Naumovich and George S. Avrunin. A conservative data flow algorithm for detecting all pairs of statements that may happen in parallel. In *Proceedings of the Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 24–34, November 1998.

[NAC99a] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. Data flow analysis for checking properties of concurrent Java programs. In *Proceedings of the* 21^{st} *International Conference on Software Engineering*, pages 399–410, May 1999.

[NAC99b] Gleb Naumovich, George S. Avrunin, and Lori A. Clarke. An efficient algorithm for computing MHP information for concurrent Java programs. In *Proceedings of the Seventh European Software Engineering Conference held jointly with the Seventh ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 338–354, September 1999.

[NACO97] Gleb Naumovich, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. Applying static analysis to software architectures. In *Proceedings of the Sixth European Software Engineering Conference held jointly with the Fifth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 77–93, September 1997.

[NCC99] Gleb Naumovich, Lori A. Clarke, and Jamieson M. Cobleigh. Using partial order techniques to improve performance of data flow analysis based verification. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 57–65, September 1999.

[NCO98] Gleb Naumovich, Lori A. Clarke, and Leon J. Osterweil. Efficient composite data flow analysis applied to concurrent programs. In *Proceedings of the ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering*, pages 51–58, June 1998.

[OF76] Leon J. Osterweil and Lloyd D. Fosdick. DAVE - a validation error detection and documentation system for fortran programs. *Software – Practice and Experience*, 6:473–486, 1976.

[OO90] Kurt M. Olender and Leon J. Osterweil. Cecil: A sequencing constraint language for automatic static analysis gereration. *IEEE Transactions on Software Engineering*, 16(3):268–280, March 1990.

[OO92] Kurt M. Olender and Leon J. Osterweil. Interprocedural static analysis of sequencing constraints. *ACM Transactions on Software Engineering and Methodology*, 1(1):21–52, January 1992.

[ORSSC98] Sam Owre, John Rushby, N. Shankar, and David Stringer-Calvert. PVS: an experience report. In Dieter Hutter, Werner Stephan, Paolo Traverso, and Markus Ullman, editors, *Applied Formal Methods—FM-Trends 98*, volume 1641 of *Lecture Notes in Computer Science*, pages 338–345, Boppard, Germany, October 1998. Springer-Verlag.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Eighteenth Symposium on Foundations of Computer Science*, pages 46–57, October 1977.

[Tay83] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19(1):57–84, April 1983.

[TBC+88] Richard N. Taylor, Frank C. Belz, Lori A. Clarke, Leon Osterweil, Richard W. Selby, Jack C. Wileden, Alexander L. Wolf, and Michal Young. Foundations for the Arcadia environment architecture. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 1–13, November 1988.

[WVF95] Jeannette M. Wing and Mandana Vaziri-Farahani. Model checking software systems: A case study. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–139, October 1995.