FLAVERS: A finite state verification technique for software systems

by J. M. Cobleigh
L. A. Clarke
L. J. Osterweil

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.

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, regression testing, requirements-based testing, and stress testing. Although all these activities serve a valuable purpose, in general, none can ensure 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 nondeterminacy. When there is nondeterminacy, the same test case may produce different results on different executions. Thus, testers of distributed systems cannot be sure that a software system will continue to produce correct results for previously successful test cases. Moreover, it may be difficult to reproduce erroneous results with test cases that exposed failures on previous executions.

The limitations of testing have long been recognized.[1] Formal verification techniques, based on theorem proving, were originally proposed to counter some of these limitations.[2] Formal verification techniques attempt 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,[3,4] 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 counterexample, 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 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, whereas 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 (Symbolic Model Verifier)[5,6] and SPIN,[7] 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 in which analysts have successfully represented and verified software systems using these approaches[8,9] 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.[10] 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, FLow Analysis for VERification of Systems, has been developed for application to the analysis of systems written in the Ada or Java** languages. The initial experiences we have had with FLAVERS, described in this paper, suggest that there are several reasons why this 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 syntactically 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.

The next section provides a high-level overview of the FLAVERS approach. The following section presents an example, first for a single task and then for a multithreaded system. Following sections present some experimental results showing how FLAVERS scales to handle larger systems and a short description of related work. Finally, the conclusion summarizes the contributions of this approach and discusses future research directions.

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 they contrive to select test data for which 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 example in the next section illustrates the process of using FLAVERS to verify properties and some of the artifacts that are created in doing so.

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. Although this may appear to be overly restrictive, in practice there are many examples of important properties that are expressible in 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 occurred. Because this variable had previously been set to true due to the firing of the retro rocket, 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 an FSV 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, $F = (S, \Sigma, \delta, s^0, A)$ 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 a 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.[11] 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 undesirable) property or to verify that *all* executions must always necessarily satisfy a (presumably desirable) property. Specification of the 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.

Modeling the system. FLAVERS analysis requires the creation of an abstract graph model of the system being analyzed. The model is a generic, language-independent representation that depicts all possible sequences 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[12,13] or Java[14] code. We have also used manual translations to demonstrate the feasibility of building such models for a high-level architectural description language[15] and a process definition language.[16]

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., Aho,[17] 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.[18] In the TFG, all method invocations have been replaced by expansions of the methods 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 overapproximates 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, although any sequence of events that can occur during an actual execution will be represented by a path in the TFG, 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 overapproximation, 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.[19] 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 overapproximate 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 using a *may happen in parallel* (MHP) analysis,[20,21] but, as previously noted, may overapproximate 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 $G = (N, E, n_{\text{initial}}, n_{\text{final}}, \Sigma_G, L)$ where N is a finite set of nodes, $E \subseteq N \times N$ is a set of directed edges, n_{initial}, $n_{\text{final}} \in N$ 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.

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 *constraints*. Like properties, 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 entered whenever the semantic feature they embody is violated. Thus, for example, a constraint FSA may be created and used to identify paths for which execution would require that a particular variable have the values 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 them determine what parts of a system need to be modeled more precisely.

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.[22] Suppose we wish to verify a property $P = (S_P, \Sigma_P, \delta_P, s_P^0, A_P)$ over a TFG $G = (N, E, n_{\text{initial}}, n_{\text{final}}, \Sigma_G, L)$ using a set of constraints $C_1, \ldots, C_k$ where $C_i = (S_{C_i}, \Sigma_{C_i}, \delta_{C_i}, s_{C_i}^0, A_{C_i}, v_{C_i})$. 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

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

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.[23] 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. Because we are generally concerned only with terminating program executions, only the tuples on n_{final} are examined. The tuples on the final node are all of the combinations of the states of the property and the states of the constraints that occur on terminating program executions. We look for violating tuples on the final node. A *violating tuple* is one for which the property automaton is in a nonaccepting state, representing a property violation, and for which every constraint FSA is in an accepting state, ensuring that all constraints are satisfied. More formally, a violating tuple is $t = (s_P, s_{C_1}, \ldots, s_{C_k})$ where $\forall 1 \le i \le k : s_{C_i} \in A_{C_i}$ 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, 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

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

Using the formal definitions, we can provide a state propagation algorithm to verify a property P over a TFG G with constraints $C_1, \ldots, C_k$. Following is the state propagation meta-algorithm (MA).

Initially:

$$Wlist := n_{\text{initial}}$$

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

Main loop:

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

The state propagation algorithm 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).[24] 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, $Tuples[n_{\text{final}}]$, 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 counterexample 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 $n_1 = n_{\text{initial}}$, $n_l = n_{\text{final}}$, 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 Cobleigh et al.[25]

The FLAVERS state propagation algorithm has worst-case complexity that is $\mathcal{O}(N^2 \cdot |S|)$, where N is the number of nodes in the TFG, and $|S|$ is the product of the number of states in the property 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 subcubic, 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.

Examples

This section presents two examples that demonstrate the types of analysis that can be done using FLAVERS and shows the artifacts that are created during the analysis. Both examples prove properties about a software system for solving the dining philosophers problem, an example that has been frequently used in software analysis literature. The first example does not have concurrency and checks the property that a philosopher must have both forks to eat. The second example, a concurrent example, checks the property that adjacent philosophers cannot eat at the same time.

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 dining philosophers problem details a scenario 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 philosopher to the left and the philosopher to the right will be unable to eat until the dining philosopher finishes eating and relinquishes the forks. Our program models the situation for the specific case of two philosophers and two forks[26] (generalization to any number of forks and philosophers is straightforward):

```
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 philosopher1
       = new Philosopher(fork1, fork2);
    Philosopher philosopher2
       = new Philosopher(fork1, fork2);
    philosopher1.start();
    philosopher2.start();
  }
}
```

Note that the class Philosopher is defined to be a thread for which 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.[27]

The class Fork implements the two methods, up and down, which 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 philosopher1, one instance of class Philosopher, and then on philosopher2, 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 run-time 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 another 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 is not able to eat until both adjacent forks have been acquired and raised to the up position. While our cursory description of our simulation 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 instance will have both Fork instances that were bound as parameters placed into the isUp state. We thus need a property FSA representing the desired up or down status of the two forks. To support the reasoning needed to assure this, we model the acquisition, or raised status, of a philosopher's adjacent forks by a property FSA shown in Figure 1.[28]

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 1 Property "has-two-forks-to-eat"

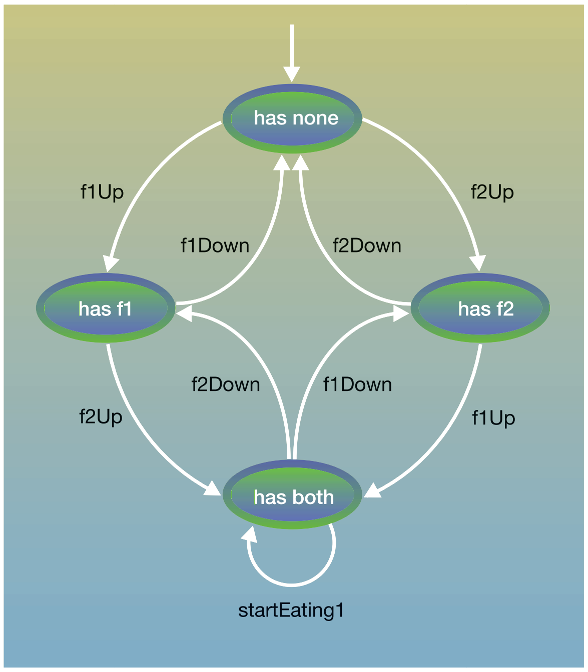
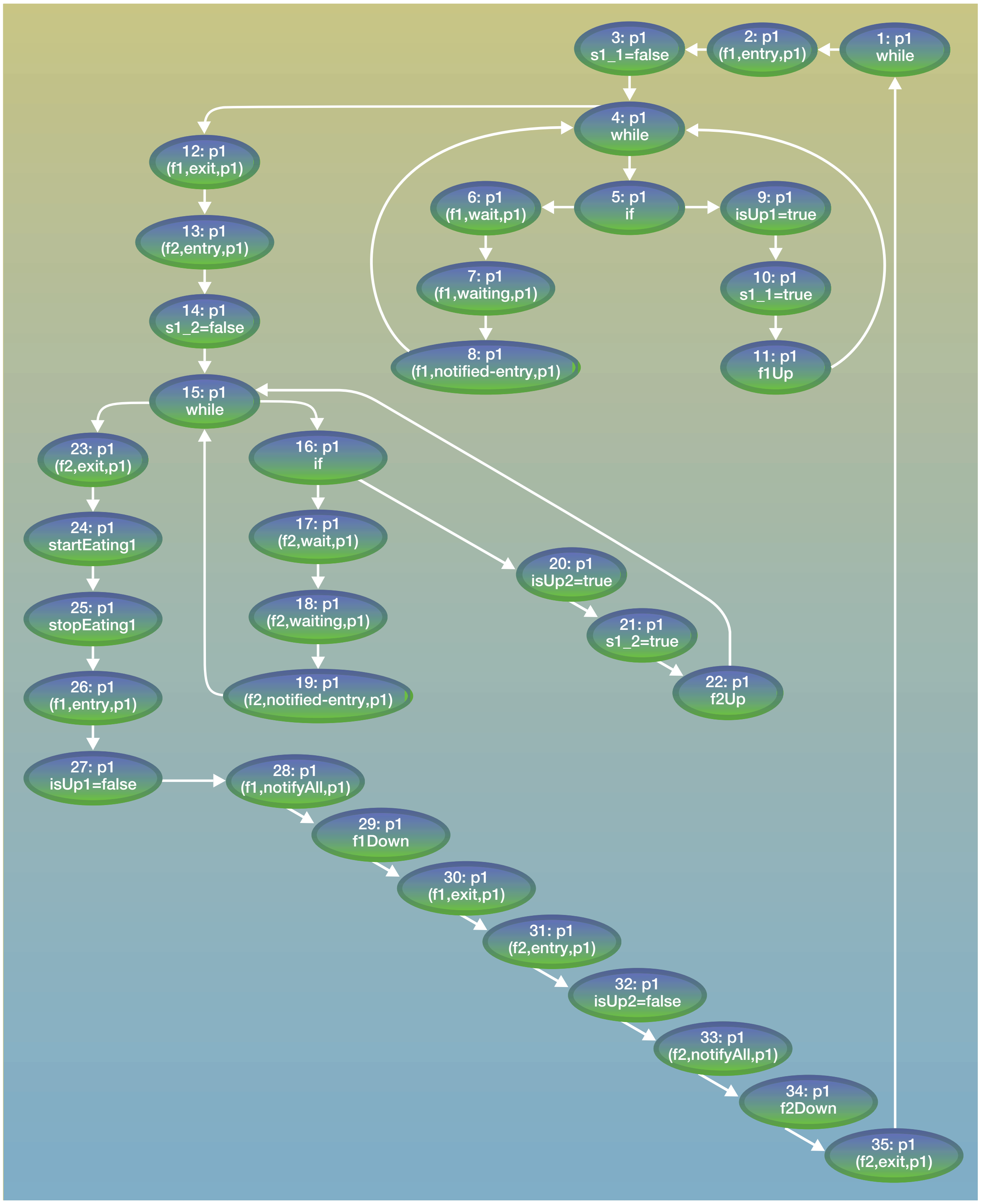


Figure 2 is an example of a control flow graph (CFG) model after this inlining process has been applied. This particular graph represents the inlining of the method invocation philosopher1.start( ) from method Main. The ovals correspond roughly to the statements that are executed as a consequence of this method call. Each oval is annotated with two lines. The top line is a unique node identifier followed by the name of the thread. The second line represents the type of action effected by the execution of the statement represented by this node. For brevity, we shorten names, so philosopher1 is replaced by p1, fork1 by f1, and success by s. Thus, note that node 4 represents execution of a while statement. Accordingly note that there are two out edges from this node, one representing looping and the other loop termination. Nodes 2, 6, 7, and 8 represent the execution of different synchronization functions. They model philosopher1 acquiring the lock on fork1, executing a

Figure 2 Control flow graph for philosopher1.run ()



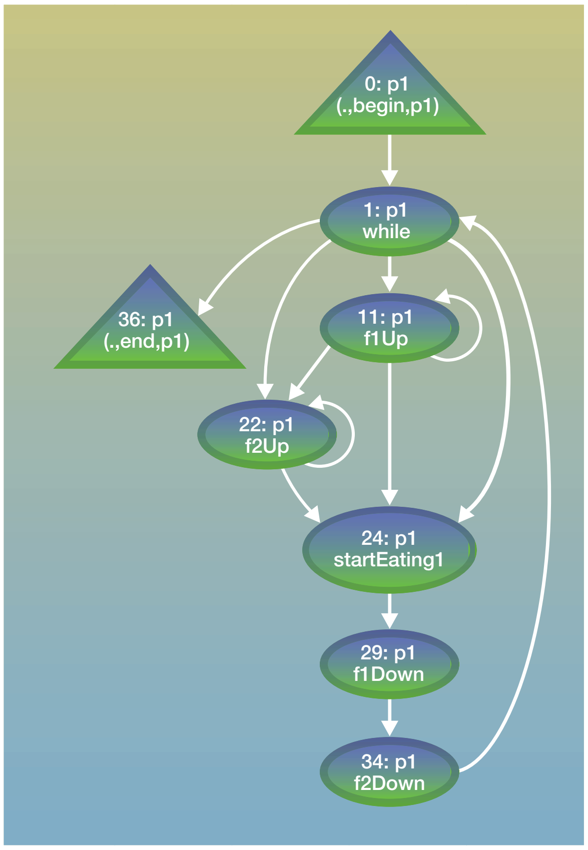
wait statement on fork1, idling while waiting to be notified, and reacquiring the lock after being notified, respectively. These nodes are not particularly important to the analysis currently being described. Their importance will be addressed in the context of a subsequent example. Finally, note that node 11 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 this FSA take place in statements represented by node 22 (the f2Up event), 24 (the startEating1 event), 29 (the f1Down event), and 34 (the f2Down event). It is particularly noteworthy that relatively few nodes in this CFG carry annotations found in the FSA. Thus, Figure 3 represents the TFG that results from reducing the CFG in Figure 2 to reflect only events used in the has-two-forks-to-eat property FSA. This TFG has two additional nodes, the initial and final nodes, nodes 0 and 36, respectively, that represent the program start and end points.

Having specified the property 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 inconclusive results, returning the path $0 \rightarrow 1 \rightarrow 24 \rightarrow 29 \rightarrow 34 \rightarrow 1 \rightarrow 36$ as a violating path, namely a path for which execution would leave the FSA in a nonaccepting state. Looking back at the original CFG in Figure 2, we can see that the only way to get from node 1 to node 24 while avoiding node 11 involves exiting the loop headed by statement 4 (the representation of the statement while(! success1_1)) without setting success1_1 to true.[29] 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 3, and it is not reset to true anywhere except in a statement represented by node 10. But it is easy to see how FLAVERS will not recognize this, because the TFG in Figure 3 does not represent those specific program details. As noted, such details are suppressed in the interests of reducing graph size and potentially reducing execution time. But, as noted previously, these efficiencies can lead to loss of analytic accuracy.

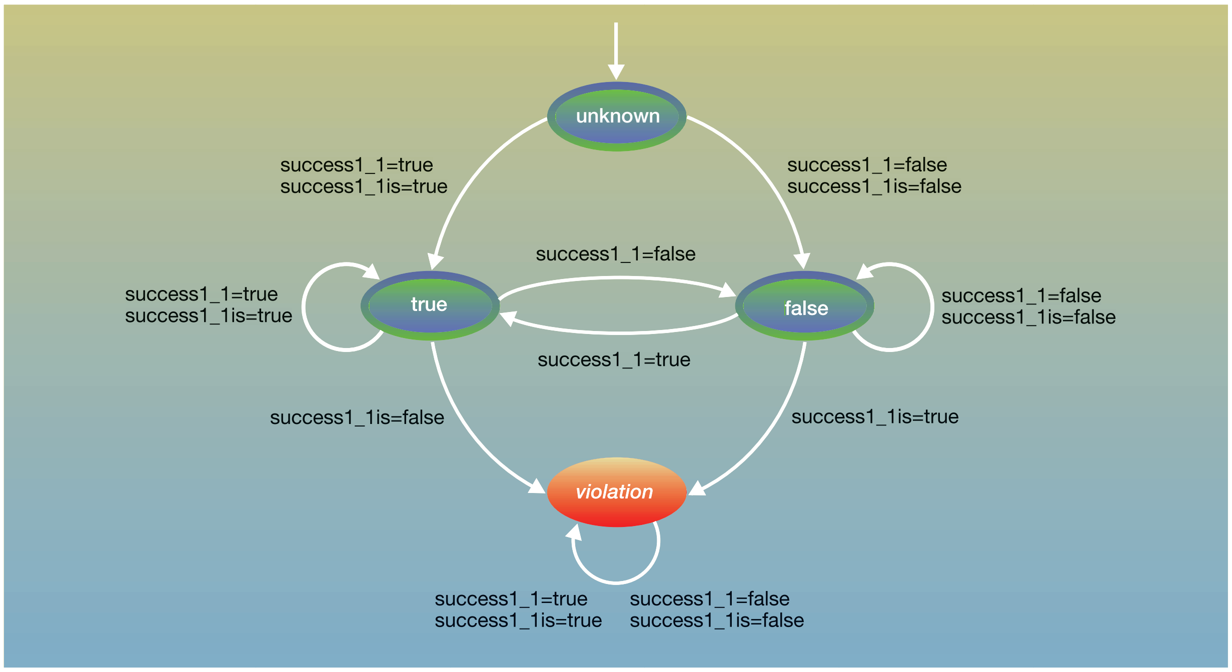
As we have stated, constraints provide a way to incrementally improve accuracy by supporting the removal of infeasible paths from consideration. This is done by introducing constraint FSAs, consideration of which increases graph size and can increase execution time, but also increases analytic precision. In this case, we want to model the value of the variable success1_1 by the constraint FSA shown in Figure 4.

Figure 3 Trace flow graph for example



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 = true and success1_1 = false, and two representing tests of the value of the variable that return the two possible different values, success1_1is = true and success1_1is = false. 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

Figure 4 Variable automaton for “success1_1”



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 causes this constraint automaton to enter 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 = false 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.

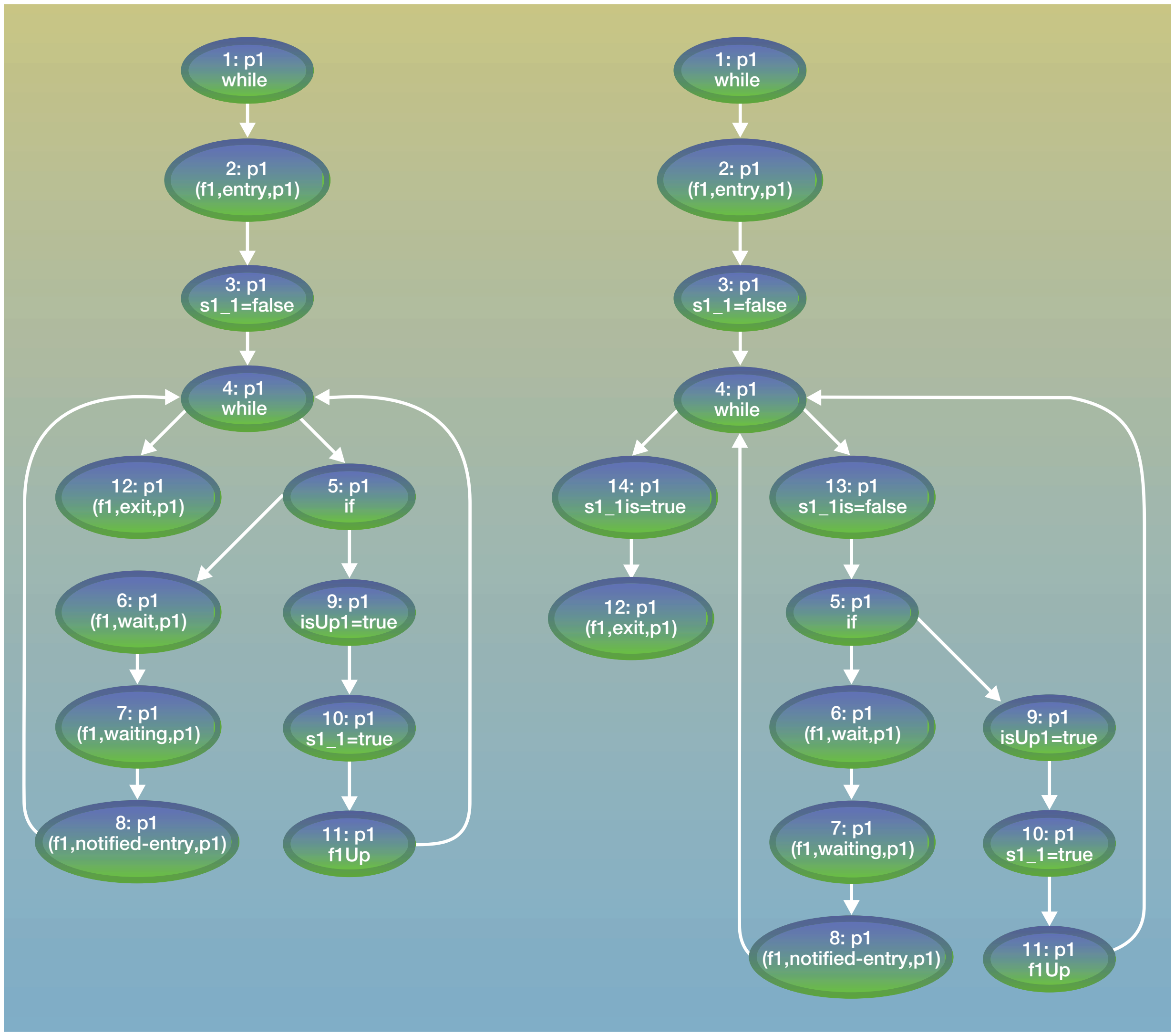
As noted previously, when the FLAVERS state propagation algorithm determines that a constraint automaton has entered a violation state it uses this information to decline to consider event sequences along further extensions of this path. For FLAVERS to use this approach to sharpen the accuracy of its analysis of the program shown earlier, 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 4 occur.

Figure 5 shows a fragment of the CFG from Figure 2 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 the augmented TFG that would result from using the augmented CGF, 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.

Example requiring analysis of concurrency. Our first example property could be studied by analyzing the code for only one thread. There will be some

Figure 5 Adding variable events to control flow graph

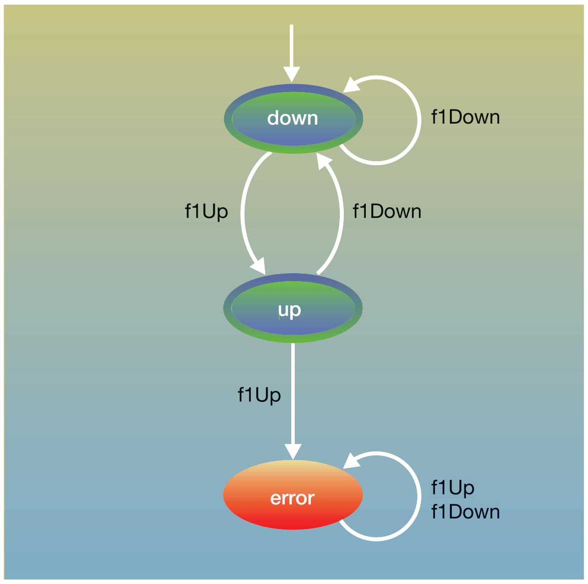


properties of this sort in a concurrent program, but quite often it will be the case that some important properties can be studied only by taking into account the combined effects of different threads of control.

As an example we now consider the property no-fork-raised-twice, namely that no fork could ever possibly be raised twice in succession. 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 6 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.

The violation of interest, raising a fork more than once, might occur if a given fork is raised from each of two different threads of control. Thus considering only one thread of control, as was done in the previous example, will not represent the full range

Figure 6 Property "no-fork-raised-twice"



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 graphs that are then interconnected by MIP edges. We illustrate this by developing the TFG needed to verify this property. Figure 7 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 7 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 7. 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 29. 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[20,21] that computes these pairs automatically based upon graphs annotated very much as shown in Figure 7. Once the sequences 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.[30] 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 8 we have used dashed lines to show only the set of MIP edges that are incident with node 47.

Having developed the TFG containing 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 counterexample 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 are connected by a MIP edge, indicating that this interleaving is possible. Examination of this counterexample path reveals that it is 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.

Figure 7 Start of TGF for “no-fork-raised-twice”

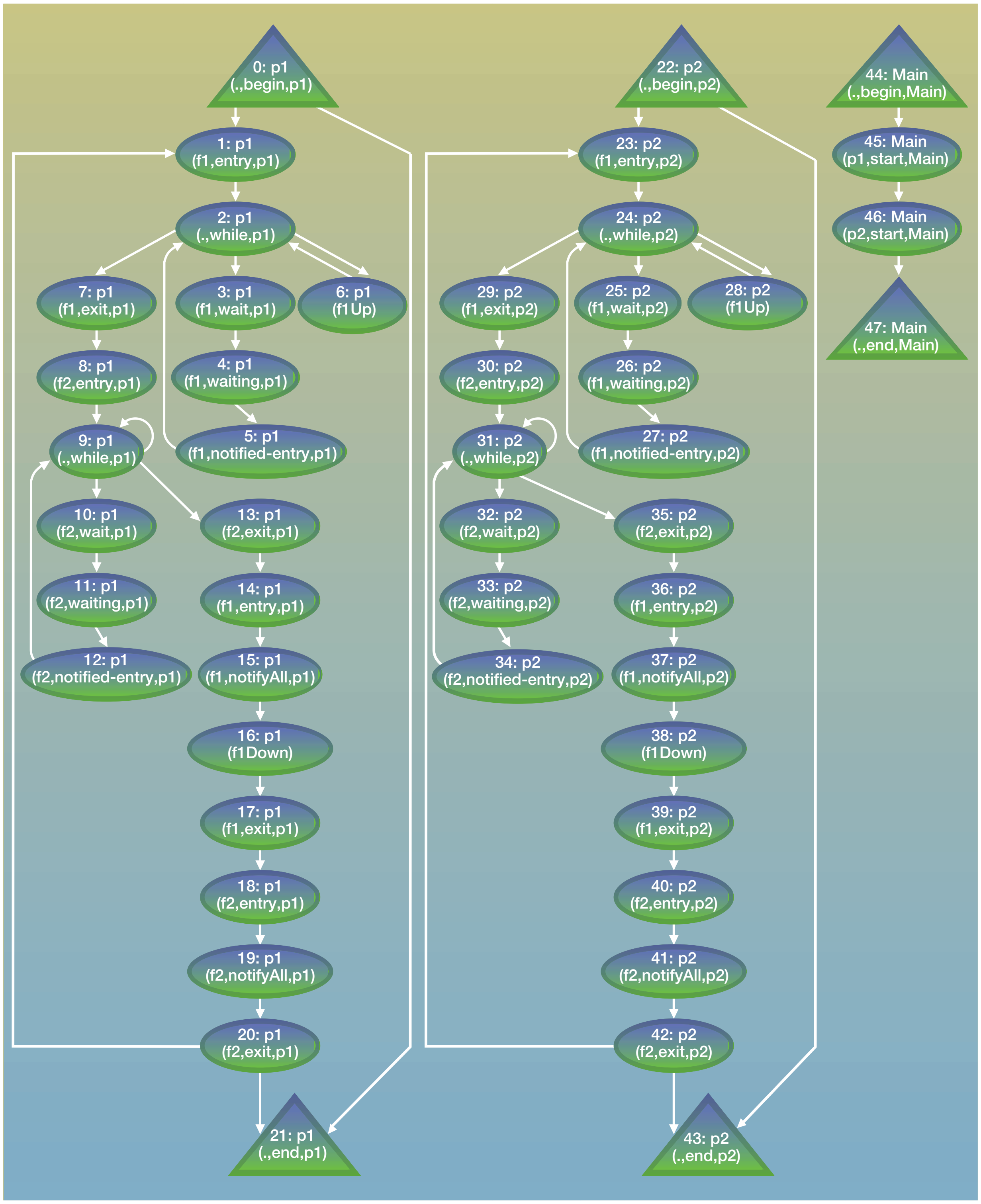
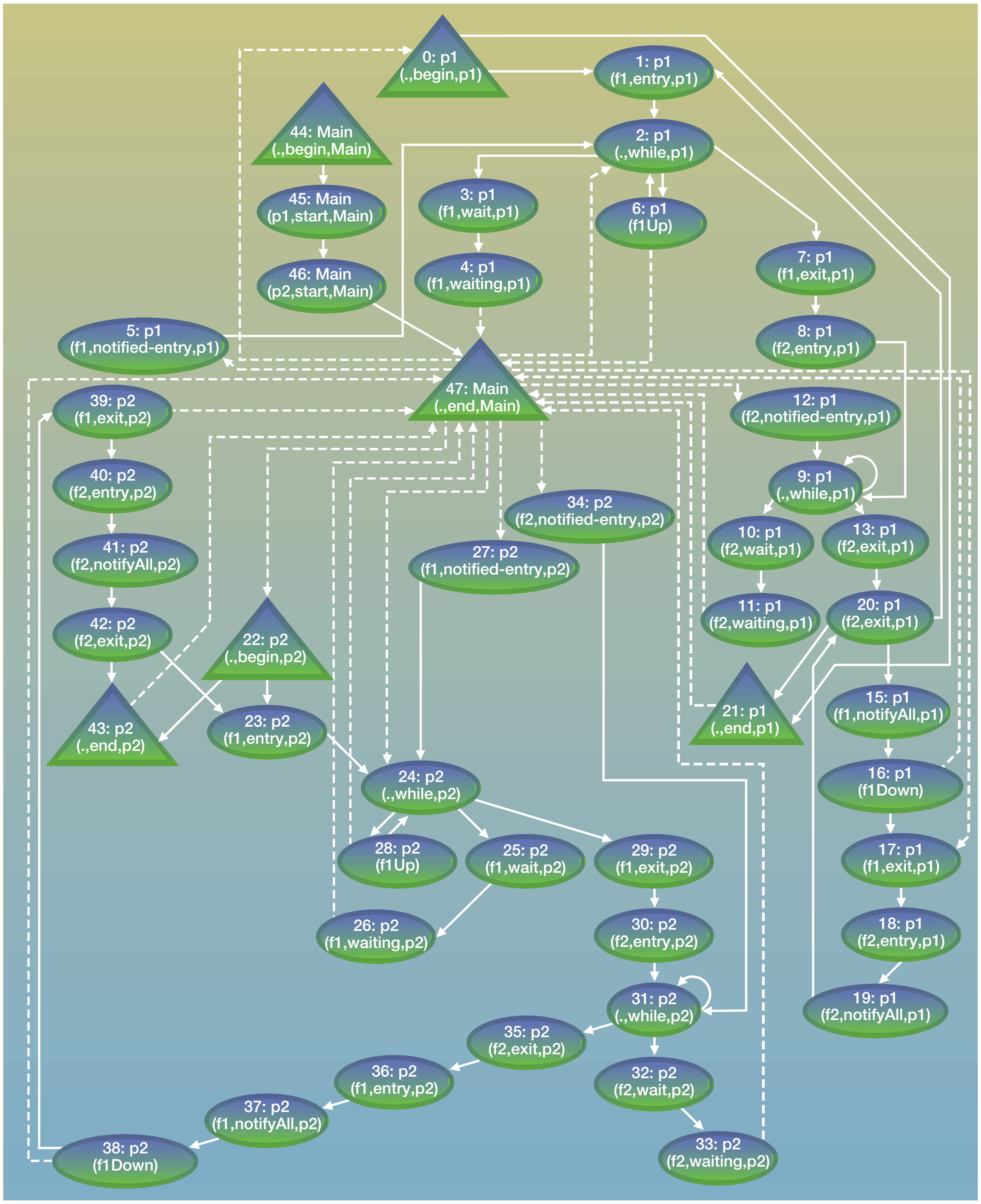


Figure 8 TGF for “no-fork-raised-twice” with MIP edges for node 47

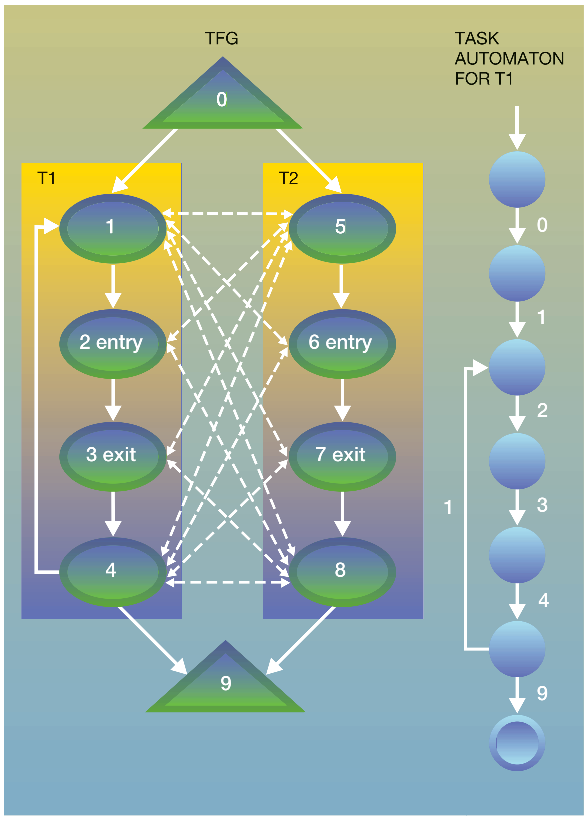


Having incorporated this variable automaton into the analysis, FLAVERS still returns an inconclusive result, in this case, producing a counterexample that includes a subpath that goes from node 6, the node in philosopher1 with event f1Up, to a node in another thread via a MIP edge, and then immediately 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 constraint, referred to as a task automaton. The purpose of a task automaton for a particular path is to ensure 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 shown in Figure 9 and assume that nodes 2, 3, 6, and 7 all access the same lock. The task automation for task T1 is shown on the right-hand side of Figure 9. The alphabet of the task automaton is the set of identifying numbers for the nodes in the TFG. Without this task automaton, FLAVERS would consider the path $0 \rightarrow 1 \Rightarrow 8 \rightarrow 9$, which is infeasible because in task T1 node 2 must follow node 1. The subset of this path that occurs in task T1, namely $0 \rightarrow 1 \rightarrow 9$, would be rejected by the task automaton since node 2 must follow node 1.

When we perform the analysis with the task automaton for the philosopher1 thread, we get inconclusive results. The counterexample 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 constraint to ensure the lock is respected (the template for this constraint is shown in Figure 10), FLAVERS returns conclusive results for the property no-fork-raised-twice.

These two properties are representative of the kind of properties that can and should be verified to show

Figure 9 Task automaton example



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 11. 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 philosopher1 and philosopher2 and a monitor constraint for fork2. These six constraints are sufficient to prove this property for this system with any number of philosophers.

Some other example properties are: a fork cannot be put down unless it is already up, a fork that is down

Figure 10 Monitor constraint

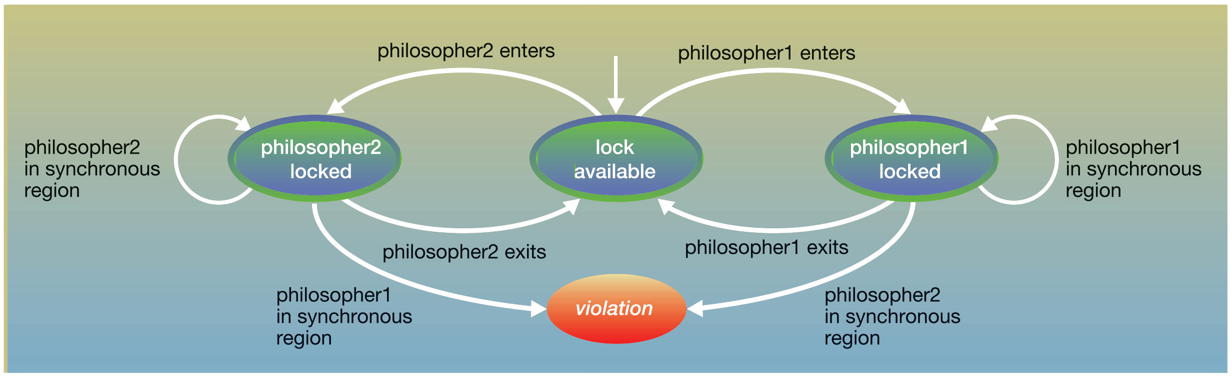
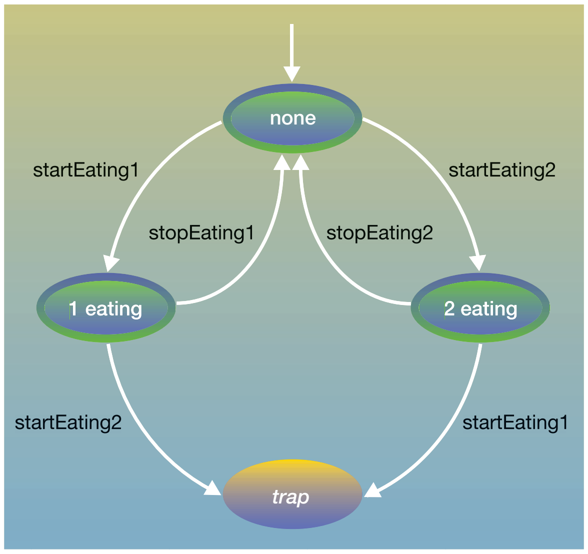

philosopher2 enters
philosopher1 enters
philosopher2 in synchronous region
philosopher2 locked
lock available
philosopher1 locked
philosopher1 in synchronous region
philosopher2 exits
philosopher1 exits
philosopher1 in synchronous region
violation
philosopher2 in synchronous region

Figure 11 Property "no-eat-at-same-time"



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.

Note that task automata and concurrency control FSAs can easily be automatically derived from the TFG. 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 for making reasonable choices automatically. Moreover, for explanatory purposes, we have selected constraints one at a time. In practice, analysts tend to add a small number of constraints at each iteration. For example, in the sequential example shown previously, we added each variable automaton separately, while an astute analyst would have suspected that both automata were important to the property being proven. In fact, an analyst might have suspected this from the very start and modeled this information in the initial analysis.

Some experimental results

Our FLAVERS prototype for analyzing programs written in Ada is considerably more mature than the prototype for Java code. Therefore all the results presented in this section are based on FLAVERS/Ada. The Ada version of our example problem for *n* philosophers has 2*n* tasks, one task for each philosopher and one task for each fork. In this section we present the timing results for analyzing the dining philosophers 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.

It should be noted that observing how size and running speed grow with increasing numbers of identical tasks may not accurately predict growth in size and speed as systems grow in less regular ways. But this approach, although flawed, does give some sense of scalability. Further, we note that others have argued that in most cases it is sufficient to validate only a small number of configurations[31] and we basically agree.

To prove the property no-fork-raised-twice conclusively for the Ada version of the program only required a task automaton for the fork1 task. To prove the property no-eat-at-same-time conclusively for the Ada version of the program required three task automata, one for the first philosopher, one for the second philosopher, and one for 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 constraints needed to prove this property conclusively. We have found that this is often the case. Once the necessary 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 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 Taylor et al.[32] Tools written in the Java language are then used to translate these CFGs into a TFG and to construct finite state automata representations of the properties and any 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 code.

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 336 megahertz processors and 2 gigabytes of memory running Solaris** 2.6. While this is a multiuser system, for all experiments we had exclusive access to the machine to prevent variance in the times due to other users. The Ada portion of the FLAVERS/Ada tools were compiled using the Verdix Ada Development 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 1.3.0. 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 12 and 13. The x-axis shows the number of philosophers and the y-axis shows the running time in seconds. Each of these figures has three lines. The sp_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. The mip_total includes these times, plus the time for running the MHP analysis and adding the MIP edges.

We are not including the time for running the Ada tools to translate the source code into annotated CFGs, since these tools are based on an obsolete front end. This translation took almost one hour when the system had 200 philosophers, whereas everything else took less than 4 minutes.

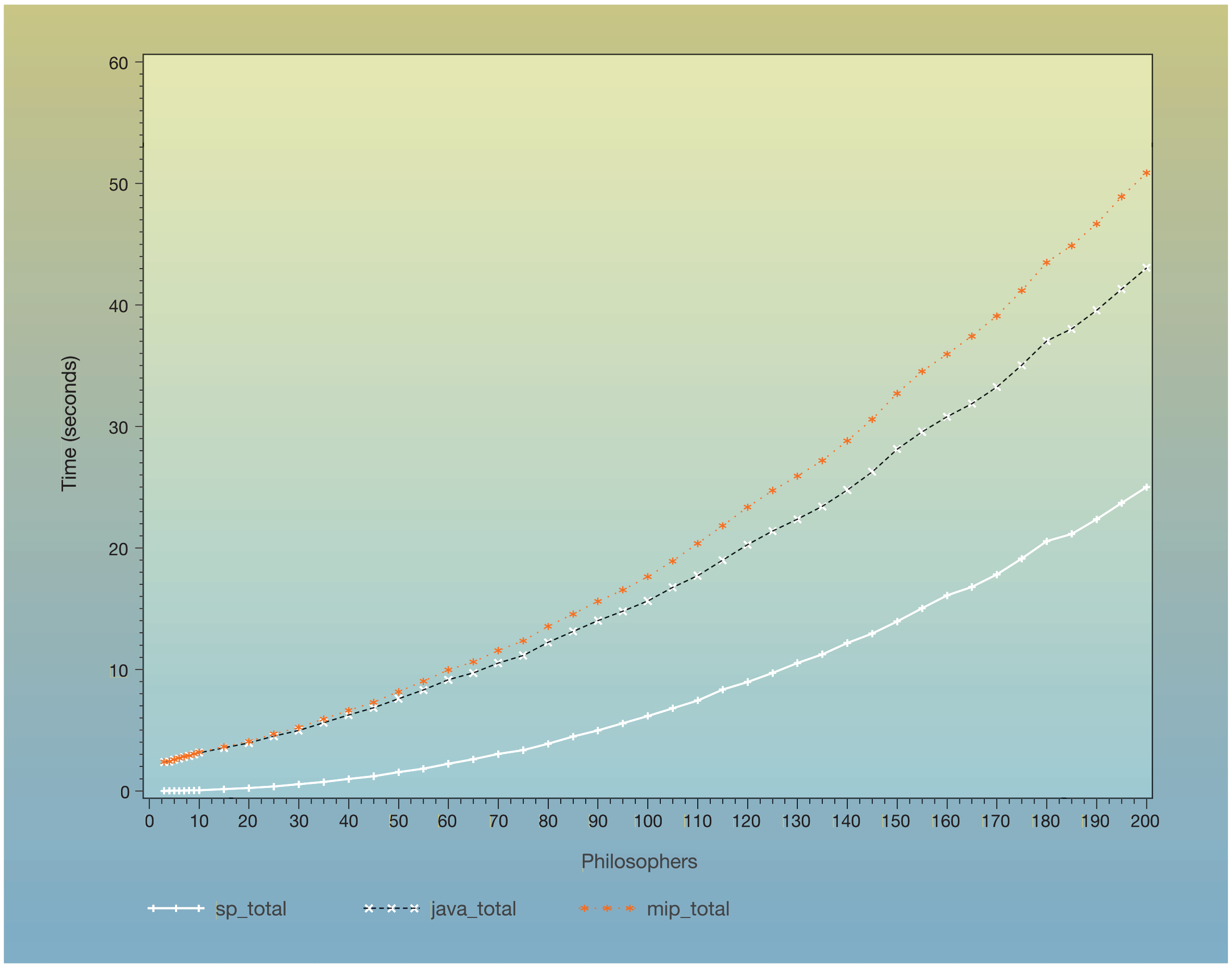
In an attempt to estimate the actual functional dependence between running time and the number of philosophers, we fit different polynomials to the mip_total lines shown in Figures 12 and 13. The results of these fittings are shown in Table 1. Each column gives the data for the best-fit polynomial of the given form. In each column, r^2 is the proportion of the variance in the data explained by the polynomial. The remaining rows give the coefficients of the best-fit polynomial.

For each property, the more terms in the polynomial, the better the fit as measured by r^2. For both properties, a linear polynomial explains the data well, but adding a quadratic term improves the fit. Adding a cubic term improves the fit, but not by much. Since the linear polynomial explains a large part of the variance, it is hard to say by examining the table whether the data are better explained by a linear or a quadratic polynomial. Figure 14 shows the total time and the linear and quadratic best-fit polynomials for the property no-fork-raised-twice. The data for the property no-eat-at-same-time is similar. From this, it appears that the data are better explained by a quadratic polynomial.

Admittedly, the dining philosophers problem 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 Avrunin et al.,[33] a comparison was made of several finite state verification tools on a real software system, called Chiron.[34] In that study, SPIN,[7] SMV,[6] INCA (Inequality Necessary Condition Analyzer),[35] and FLAVERS were all applied to the same problems.

Figure 12 Timing for “no-fork-raised-twice”



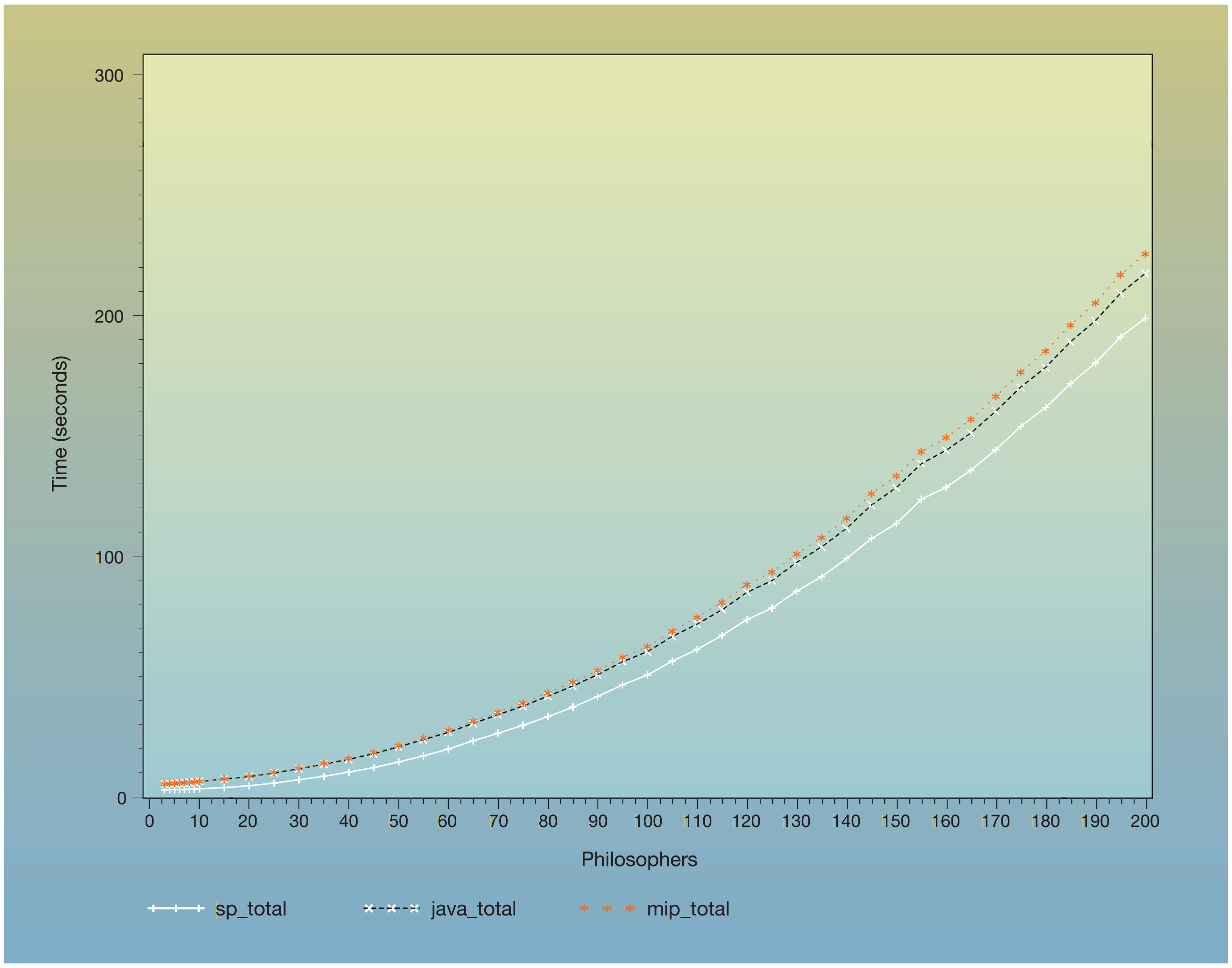
Although there was considerable variation in how the tools performed from problem to problem, the computation time associated with FLAVERS, with a few exceptions, grew at a lower rate than that associated with the other tools.[36]

Related work

Flow analysis techniques for verification were originally used to detect potential definition and reference anomalies in sequential code.[37] This approach was later extended to allow the verification of user-specified properties.[11,38] FLAVERS extended this work to support verification of concurrent systems and to support incremental improvements to the model of the system being analyzed.[12,13]

Most other approaches to FSV 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. These approaches often use abstractions, such as binary decision diagrams,[6] and optimization, such as partial order reductions,[39] to reduce the size of the system model. These optimizations and abstractions have effectively reduced the model size impressively for a number of examples. When the model size is too large to support efficient analysis, however, analysts must devise new abstractions to reduce the model further. FLAVERS, in contrast, starts with a small model but provides a systematic way of improving the precision of the system model steadily with a gradual increase in model size. On the other hand,

Figure 13 Timing for "no-eat-at-same-time"



since the FLAVERS model is event-based and imprecise, it is not effective for detecting properties such as deadlock, unlike many of the reachability-based approaches.

There has been considerable interest in applying FSV approaches to software systems. Thus, for example, SPIN[7] creates a highly optimized representation of the reachability graph. Initially SPIN supported the analysis of systems written in Promela, a state transition modeling language. Recently, a translator from C to Promela (Process Meta Language) has been developed, so that C programs can be analyzed as well.[40] This translation is only partially automated, however, and requires the user to specify mappings between the C and the Promela code. For example, communication between threads may need to be mapped into communication via channels in Promela.

Java PathFinder (JPF) uses a reachability-based approach to verify properties of Java programs.[41] JPF operates directly on the Java bytecode and tries to execute all possible paths. To combat the state explosion problem, it makes use of abstraction, slicing, and partial order reduction.

The SLAM (software specifications, languages, analysis, and model checking) project verifies properties over sequential programs, using a reachability graph that represents all variables in terms of Boolean values.[42] The tool translates a C program into a Boolean program, abstracting values with respect to a set

Table 1 Polynomial fitting

	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	0.9640	0.999823	0.999826
c_0	−1.9378	2.3187	2.2711
c_1	0.2335	0.0702	0.0741
c_2		$8.660 * 10^{-4}$	$8.141 * 10^{-4}$
c_3			$1.780 * 10^{-7}$
	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	0.9359	0.999883	0.999884
c_0	−23.3302	2.3601	2.5072
c_1	1.0393	0.0540	0.0420
c_2		0.0052	0.0054
c_3			$-5.480 * 10^{-7}$

of predicates, and then uses interprocedural data flow analysis to associate variable states with the program statements.[43] When a counterexample is found, SLAM makes use of a theorem prover to detect whether the counterexample is infeasible and to find a set of predicates to remove the infeasible path from consideration. At this point, the process begins again with the translation into a new Boolean program. This iterative refinement approach is very similar to the iterative approach that we are advocating with FLAVERS.

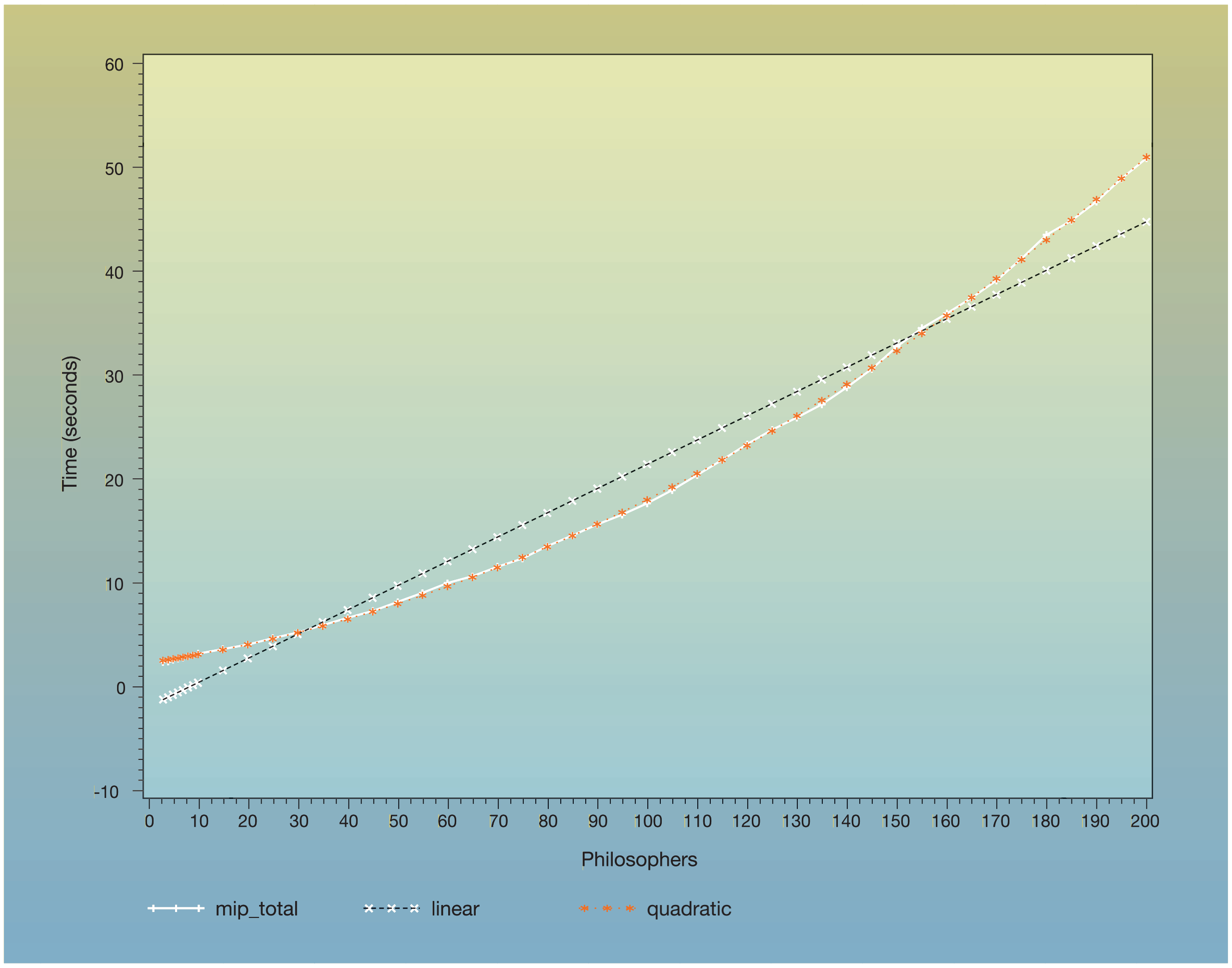
INCA[35] is an FSV approach that is not based on a reachability graph model. INCA creates inequalities that describe the legal flow through the system and models the complement of the property as an inequality as well. It then uses integer linear programming techniques to determine if there exists a solution representing flow through the system that is a violation of the property. Although, in general, integer linear programming has an exponential worst-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.

The Bandera project[44] is concerned with automatically extracting system models from Java programs. Bandera uses abstraction[45] and slicing techniques[46] to reduce the size of the model. We are currently exploring the use of the Bandera tool set to build FLAVERS' TFGs for Java programs. In doing so, we believe that the slicing and abstractions will provide a good starting point for doing FLAVERS analyses, but that slicing will tend to overestimate the relevant portions of the programs that are necessary for proving that a property holds. As a result, we expect that it will be necessary to allow the analyst to add precision more selectively to the model using FLAVERS' constraint mechanism.

In addition to FSV approaches, which verify properties for all executions, there are some interesting approaches that generate counterexamples to a property. These approaches are not conservative and thus are not guaranteed to detect a violation if one exists. For example, the Alloy constraint analyzer has been used to analyze programs written in a subset of the Java language by translating them into the Alloy language.[31] Unlike FSV tools, the counterexamples returned by Alloy are guaranteed to be feasible. The MC (metalevel compilation) system[47] checks C programs for adherence to properties specified in *metal*, a state machine language. The system is built as a compiler extension and transitions in the state machine are syntactic patterns in the source code. The MC approach is similar to FLAVERS' state propagation, in that states of the property are propagated to statements of the program being checked. Like Alloy, MC analyses find counterexamples, but do not guarantee the absence of possible property violations.

FLAVERS uses a graph model of the software system that is considerably less precise than the FSV approaches described previously. Reachability graph models represent all the states that a system can be in, where a state consists of the program counter for each task and the values of each of the task's variables. The optimization techniques employed by

Figure 14 Best-fit polynomials for “no-fork-raised-twice”



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, with respect to the property, but not precise. 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 real-world 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 much of this guidance. Based on our scaling experiments, FLAVERS seems to have a growth rate that is usually subcubic in the number of tasks. This compares very favorably with other FSV approaches we have studied.

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 widespread 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 analyses on a very precise model of the system and have thus focused 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 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 constraints, to add more detail. Thus, the model is built up incrementally, with some support for the automatic creation of commonly used types of constraints. Although the analyst has to help select the constraints to be added, we believe that this is usually more straightforward than developing the abstractions needed to reduce the size of reachability-based models. In either case, more automated techniques would be useful to assist analysts with these tasks.

Although constraints are a very powerful mechanism for selectively adding more semantic information and for modeling additional information, when they are manually created by an analyst, they may be incorrect. It is important to document the assumptions (i.e., the constraints) under which each property is verified. For those constraints that are manually created, assertion monitoring techniques might also be useful to determine if such constraints are ever violated during execution.

Our experimental results indicate that the FLAVERS approach is effective for verifying a wide range of event-based properties. In addition to the dining philosophers system, we have verified properties for a number of Ada programs. 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 has 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++. In addition, FLAVERS has been used to verify properties of process programs[16] and architecture descriptions.[15]

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, ranging from one hundred lines to thirty thousand lines of source code. Unfortunately, the cost of such verification seems to be very dependent on the actual system and properties being considered.[48] Systems that have events on most statements and have a great deal of intertask communication tend to require considerably more time and space for their analysis. Moreover, small changes in a program or in a property can have a large impact on resource utilization, causing predictions of the cost for a particular analysis to be unreliable.

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 Java concurrency constructs,[14] more experimentation needs to be done.

Many of the optimizations used in FLAVERS are, or could be, applied by other FSV systems. The Bandera system is trying to provide general front-end support, for FSV systems, that would incorporate many of the transformations used in FLAVERS, as well as several others. Others are exploring alternative reduced internal representations. Although these optimizations and reduced representations help in many cases, they do not appear to be adequate when dealing with most software systems. FLAVERS' incremental approach assumes that, even when such optimizations are applied, the model will usually be unnecessarily large and thus it is preferable to produce a smaller, more tractable model initially and then rely on constraints to improve the model. Currently, the analyst must make the decisions about what constraints to add, but in our future work we intend to explore ways of providing more automated support for constraining the problem based on past results.

In addition, we are also 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-based notations,[49,50] it is still difficult to capture a property specification precisely. Building upon the work in specification patterns,[51] 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.

**Trademark or registered trademark of Sun Microsystems, Inc.

Cited references and notes

1. E. W. Dijkstra, *Notes on Structured Programming*, Academic Press, London (1972), pp. 1–82.
2. R. Floyd, "Assigning Meaning to Programs," *Proceedings, Symposium on Applied Mathematics*, Volume 19, American Mathematical Society, Providence, RI (1967), pp. 19–32.
3. S. Owre, J. Rushby, N. Shankar, and D. Stringer-Calvert, "PVS: An Experience Report," *Applied Formal Methods—FM-Trends 98*, Volume 1641 of *Lecture Notes in Computer Science*, D. Hutter, W. Stephan, P. Traverso, and M. Ullman, Editors, Springer-Verlag, Boppard, Germany (October 1998), pp. 338–345.
4. E. M. Clarke and J. M. Wing, "Formal Methods: State of the Art and Future Directions," *ACM Computing Surveys* **28**, No. 4, 626–643 (December 1996).
5. 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**, No. 2, 142–170 (1992).
6. K. L. McMillan, *Symbolic Model Checking: An Approach to the State Explosion Problem*, Kluwer Academic Publishers, Boston, MA (1993).
7. G. J. Holzmann, "The Model Checker SPIN," *IEEE Transactions on Software Engineering* **23**, No. 5, 279–295 (May 1997).
8. R. J. Anderson, P. Beame, S. Burns, W. Chan, F. Modugno, D. Notkin, and J. D. Reese, "Model Checking Large Software Specifications," *IEEE Transactions on Software Engineering* **24**, No. 7, 498–520 (July 1996).
9. J. M. Wing and M. Vaziri-Farahani, "Model Checking Software Systems: A Case Study," *Proceedings, Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Washington, DC (October 10–13, 1995), pp. 128–139.
10. This research was partially supported by the U.S. Department of Defense/Army and the Defense Advanced 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. Department of Defense, the U.S. Army, the U.S. government, the National Science Foundation, or of IBM.
11. K. M. Olender and L. J. Osterweil, "Cecil: A Sequencing Constraint Language for Automatic Static Analysis Generation," *IEEE Transactions on Software Engineering* **16**, No. 3, 268–280 (March 1990).
12. M. B. Dwyer and L. A. Clarke, "Data Flow Analysis for Verifying Properties of Concurrent Programs," *Proceedings, Second ACM SIGSOFT Symposium on the Foundations of Software Engineering*, New Orleans, LA (December 6–9, 1994), pp. 62–75.
13. M. B. Dwyer and L. A. Clarke, *Flow Analysis for Verifying Specifications of Concurrent and Distributed Software*, Technical Report 99-52, University of Massachusetts, Department of Computer Science, Amherst, MA (August 1999).
14. G. Naumovich, G. S. Avrunin, and L. A. Clarke, "Data Flow Analysis for Checking Properties of Concurrent Java Programs," *Proceedings, 21st International Conference on Software Engineering*, Los Angeles, CA (May 16–22, 1999), pp. 399–410.
15. G. Naumovich, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Applying Static Analysis to Software Architectures," *Proceedings, Joint 6th European Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Zurich, Switzerland (September 22–25, 1997), pp. 77–93.
16. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "Verifying Properties of Process Definitions," *Proceedings, International Symposium on Software Testing and Analysis*, Portland, OR (August 22–24, 2000), pp. 96–101.
17. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley Publishing Co., Reading, MA (1986).
18. FLAVERS does not currently handle recursive subprograms, although standard techniques for dealing with recursion could be extended and incorporated.
19. R. N. Taylor, "Complexity of Analyzing the Synchronization Structure of Concurrent Programs," *Acta Informatica* **19**, No. 1, 57–84 (April 1983).
20. G. Naumovich and G. S. Avrunin, "A Conservative Data Flow Algorithm for Detecting All Pairs of Statements that May Happen in Parallel," *Proceedings, Sixth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Lake Buena Vista, FL (November 3–5, 1998), pp. 24–34.
21. G. Naumovich, G. S. Avrunin, and L. A. Clarke, "An Efficient Algorithm for Computing MHP Information for Concurrent Java Programs," *Proceedings, Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, Toulouse, France (September 6–10, 1999), pp. 338–354.
22. G. Naumovich, L. A. Clarke, and L. J. Osterweil, "Efficient Composite Data Flow Analysis Applied to Concurrent Programs," *Proceedings, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Montreal, Canada (June 16, 1998), pp. 51–58.
23. T. J. Marlowe and B. G. Ryder, "Properties of Data Flow Frameworks: A Unified Model," *Acta Informatica* **28**, 121–163 (1990).
24. Clever bookkeeping can improve the efficiency of line 5 of the meta-algorithm. 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.

25. J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "The Right Algorithm at the Right Time: Comparing Data Flow Analysis Algorithms for Finite State Verification," *Proceedings, 23rd International Conference on Software Engineering*, Toronto, Canada (May 12–19, 2001), pp. 37–46.
26. In this example, both philosophers pick up fork 1 and then fork 2, 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.
27. The variable "done" is used to break the loop to allow the program to terminate, since FLAVERS currently considers only tuples that reach the final node. Although we do not address this issue in this paper, work has been done on extending our approach to prove properties over infinite executions. See G. Naumovich and L. A. Clarke, "Extending FLAVERS to Check Properties on Infinite Executions of Concurrent Software Systems," *Proceedings, Workshop on Engineering Automation for Software Intensive System Integration*, Monterey, CA (June 18–22, 2001), pp. 126–135.
28. FLAVERS requires that FSAs be total, meaning that they have a transition from every state on every event in the alphabet. The FSA in Figure 1 and the other FSAs that we show are not total in order to make them more compact and easier to understand. They can easily be made total, by adding a *trap* state that is nonaccepting 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.
29. When inlining was performed, variables in called methods were renamed to avoid collisions. Thus, in the philosopher1 task, success1_1 represents the variable success in the method up and success1_2 represents the variable success in the method down. Similarly, for the philosopher2 task, variables success2_1 and success2_2 are used.
30. We can do better than this by using a partial order optimization that 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. See G. Naumovich, L. A. Clarke, and J. M. Cobleigh, "Using Partial Order Techniques to Improve Performance of Data Flow Analysis Based Verification," *Proceedings, ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, Toulouse, France (September 6, 1999), pp. 57–65.
31. D. Jackson and M. Vaziri, "Finding Bugs with a Constraint Solver," *Proceedings, International Symposium on Software Testing and Analysis*, Portland, OR (August 21–23, 2000), pp. 14–25.
32. R. N. Taylor, F. C. Belz, L. A. Clarke, L. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young, "Foundations for the Arcadia Environment Architecture," *Proceedings, ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (November 1988), pp. 1–13.
33. G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Păsăreanu, and S. F. Siegel, *Comparing Finite-State Verification Techniques for Concurrent Software*, Technical Report 99-69, University of Massachusetts, Department of Computer Science, Amherst, MA (November 1999).
34. K. Forester, C. MacFarlane, M. Cameron, and G. Bolcer, *Chiron-1 User Manual*, Arcadia Document UCI-93-07, University of California, Irvine, CA (September 1993).
35. G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated Analysis of Concurrent Systems with the Constrained Expression Toolset," *IEEE Transactions on Software Engineering* **17**, No. 11, 1204–1222 (November 1991).
36. In this experiment, FLAVERS often had the worst performance on the smaller problems, but it was the only FSV system that began its analysis by dealing with actual source code, rather than an abstract model of source code.
37. L. J. Osterweil and L. D. Fosdick, "DAVE—A Validation Error Detection and Documentation System for FORTRAN Programs," *Software—Practice and Experience* **6**, 473–486 (1976).
38. K. M. Olender and L. J. Osterweil, "Interprocedural Static Analysis of Sequencing Constraints," *ACM Transactions on Software Engineering and Methodology* **1**, No. 1, 21–52 (January 1992).
39. P. Godefroid, "Model Checking for Programming Languages Using VeriSoft," *Proceedings, 24th ACM Symposium on Principles of Programming Languages*, Paris, France (January 15–17, 1997), pp. 174–186.
40. G. J. Holzmann, "Logic Verification of ANSI-C Code with SPIN," *Proceedings, Seventh SPIN Workshop*, Stanford, CA (August 30–September 1, 2000), pp. 131–147.
41. W. Visser, K. Havelund, G. Brat, and S.-J. Park, "Model Checking Programs," *Proceedings, Fifteenth IEEE International Conference on Automated Software Engineering* (September 2000), pp. 3–12.
42. T. Ball and S. K. Rajamani, "Automatically Validating Temporal Safety Properties of Interfaces," *Proceedings, Eighth SPIN Workshop*, Toronto, Canada (May 19–20, 2001), pp. 101–122.
43. T. Ball and S. K. Rajamani, "Bebop: A Symbolic Model Checker for Boolean Programs," *Proceedings, Seventh SPIN Workshop*, Stanford, CA (August 30–September 1, 2000), pp. 113–130.
44. J. C. Corbett, M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera: Extracting Finite-State Models from Java Source Code," *Proceedings, 22nd International Conference on Software Engineering*, Limerick, Ireland (June 4–11, 2000).
45. M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. S. Păsăreanu, Robby, W. Visser, and H. Zhen, "Tool-Supported Program Abstraction for Finite-State Verification," *Proceedings, 23rd International Conference on Software Engineering*, Toronto, Canada (May 12–19, 2001), pp. 177–187.
46. M. B. Dwyer, J. Hatcliff, and H. Zheng, "Slicing Software for Model Construction," *Proceedings, ACM/SIGPLAN Symposium on Partial Evaluation and Semantic-Based Program Manipulation*, San Antonio, TX (January 22–23, 1999), pp. 105–118.
47. D. Engler, B. Chelf, A. Chou, and S. Hallem, "Checking System Rules Using System-Specific, Programmer-Written Compiler Extensions," *Proceedings, Fourth Symposium on Operating Systems Design and Implementation*, San Diego, CA (October 23–25, 2000).
48. A. T. Chamillard, L. A. Clarke, and G. S. Avrunin, *An Empirical Comparison of Static Concurrency Analysis Techniques*, Technical Report 96-84, University of Massachusetts, Department of Computer Science, Amherst, MA (May 1996).
49. A. Pnueli, "The Temporal Logic of Programs," *Proceedings, Eighteenth Symposium on Foundations of Computer Science*, Providence, RI (October 31–November 2, 1977), pp. 46–57.

50. E. A. Emerson, "Temporal and Modal Logic," *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, J. van Leeuwen, Editor, Elsevier Science Publishers (1990), 995–1072.

51. M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-State Verification," *Proceedings, 21st International Conference on Software Engineering*, Los Angeles, CA (May 1999), pp. 16–22.

Accepted for publication September 19, 2001.

Jamieson M. Cobleigh *University of Massachusetts, Amherst, Massachusetts 01003 (electronic mail: jcobleig@cs.umass.edu).* Mr. Cobleigh received a B.S. degree in computer science and mathematics from Rutgers University and is currently a computer science graduate student at the University of Massachusetts. His thesis is investigating compositional approaches to finite state verification. He is a recipient of the WellFleet Fellowship and is a graduate research assistant.

Lori A. Clarke *University of Massachusetts, Amherst, Massachusetts 01003 (electronic mail: clarke@cs.umass.edu).* Dr. Clarke received a B.A. degree in mathematics from the University of Rochester and a Ph.D. degree in computer science from the University of Colorado. In 1975 she joined the computer science faculty at the University of Massachusetts, where she has continued to pursue research on a broad range of software engineering issues, including verification of distributed systems and distributed object technology. Dr. Clarke is a Fellow of the ACM, a recipient of the University of Massachusetts Chancellor's medal, a member of the IEEE Computer Society Publications Board, the board of directors of the Computing Research Association (CRA), and the steering committee for the International Conference on Software Engineering (ICSE). She is a former IEEE Distinguished Visitor, ACM National Lecturer, member of the NSF CCR advisory board, recipient of a University Faculty Fellowship, associate editor of *ACM Transactions on Programming Languages and Systems* and the *IEEE Transactions on Software Engineering*, and secretary/treasurer, vice-chair, and chair of SIGSOFT. She has served on or chaired numerous program committees and is general chair of ICSE 2003.

Leon J. Osterweil *University of Massachusetts, Amherst, Massachusetts 01003 (electronic mail: ljo@cs.umass.edu).* Dr. Osterweil is currently Dean of the College of Natural Sciences and Mathematics at the University of Massachusetts, where he is also a professor in the Department of Computer Science, codirector of the Laboratory for Advanced Software Engineering Research (LASER), and founding codirector of the Electronic Enterprise Institute. Previously he had been a professor in, and chair of, computer science departments at both the University of California, Irvine, and the University of Colorado, Boulder. He was the founding director of the Irvine Research Unit in Software (IRUS) and the Southern California Software Process Improvement Network (SPIN). He has been the program committee chair for the Sixteenth International Conference on Software Engineering, the Second International Symposium on Software Testing, Analysis, and Validation, the Fourth International Software Process Workshop, the Second Symposium on Software Development Environments, and both the Second and Fifth International Conferences on the Software Process. He was also the general chair of the Sixth ACM SIGSOFT Conference on the Foundations of Software Engineering. He has been a member of the editorial boards of the *ACM Transactions on Software Engineering Methods*, *IEEE Software*, and *Software Process Improvement and Practice*. He has presented keynote talks at CASE '92 in Montreal, Quality Week 2000 in San Francisco, the Inaugural Symposium of JAIST (the Japan Advanced Institute for Software Technology) in Kanazawa, Japan, and ICSE 9 (the Ninth International Conference on Software Engineering), where he introduced the concept of process programming. His ICSE 9 paper has been awarded a prize as the most influential paper of ICSE 9, awarded as a 10-year retrospective. He has consulted for IBM, Bell Laboratories, SAIC, MCC, and TRW, and SEI's Process Program Advisory Board. Dr. Osterweil is a Fellow of the ACM.