Improving the Precision of INCA by Eliminating Solutions with Spurious Cycles

Stephen F. Siegel and George S. Avrunin

S. F. Siegel is with the Laboratory for Advanced Software Engineering Research, Department of Computer Science, University of Massachusetts, Amherst, MA 01003. E-mail: siegel@cs.umass.edu

G. S. Avrunin is with the Department of Mathematics and Statistics, University of Massachusetts, Amherst, MA 01003. E-mail: avrunin@math.umass.edu

Abstract

The Inequality Necessary Condition Analyzer (INCA) is a finite-state verification tool that has been able to check properties of some very large concurrent systems. INCA checks a property of a concurrent system by generating a system of inequalities that must have integer solutions if the property can be violated. There may, however, be integer solutions to the inequalities that do not correspond to an execution violating the property. INCA thus accepts the possibility of an inconclusive result in exchange for greater tractability. We describe here a method for eliminating one of the two main sources of these inconclusive results.

Index Terms

INCA, finite-state verification, cycles, integer programming

I. I

Finite-state verification tools deduce properties of finite-state models of computer systems. They can be used to check such properties as freedom from deadlock, mutually exclusive use of a resource, and eventual response to a request. If the model represents all the executions of a system (perhaps by making use of some abstraction), a finite-state verification tool can take into account all the executions of the system. Moreover, finite-state verification tools can be applied at any stage of system development at which an appropriate model can be constructed. Such tools thus represent an important complement to testing, especially for concurrent systems where nondeterministic behavior can lead to very different executions arising from the same input data.

The main obstacle to finite-state verification of concurrent systems is the *state explosion problem:* the number of states a concurrent system can reach is, in general, exponential in the number of concurrent processes in the system. This problem confronts the analyst immediately—even for small systems, the number of reachable states can be large enough so that a straightforward approach that examines each state is completely intractable—and complexity results tell us that there is no way to avoid it completely. Every method for finite-state verification of concurrent systems must pay some price, in accuracy or range of application, for practicality.

The Inequality Necessary Conditions Analyser (INCA) is a finite-state verification tool that has been used to check properties of some systems with very large state spaces. The INCA approach is to formulate a set of necessary conditions for the existence of an execution of the program that violates the property. If the conditions are inconsistent, no execution can violate the property. If the conditions are consistent, the analysis is inconclusive; since the conditions are necessary but not sufficient, it may still be the case that no execution of the program can violate the property. INCA thus accepts the possibility of an inconclusive result in exchange for greater tractability. There are two main sources of inconclusive results. In this paper, we show how one of these, caused by cycles in finite state automata representing the components of the concurrent system, can be eliminated at what seems to be only moderate cost.

In the next section, we describe the INCA approach. Section III explains our technique for improving INCA's precision, and the fourth section presents some preliminary data on its application. The fifth section discusses some related work, and the final section summarizes the paper and discusses other issues related to the precision of INCA.

II. INCA

A complete discussion of the INCA approach, along with a careful analysis of its expressive power, is contained in [1]. In this section, we will use a small (and quite contrived) example to sketch the basic INCA approach and show how certain cycles in the automata corresponding to the components of a concurrent system can lead to imprecision in the INCA analysis. We refer readers who want more detail to [1].

A. *Basic Approach*

The basic INCA approach is to regard a concurrent system as a collection of communicating finite state automata (FSAs). Transitions between states in these FSAs correspond to events in an execution of the system. INCA treats each FSA as a network with flow, and regards each occurrence of a transition from state s to state t, corresponding to an event e, as a unit of flow from node s to node t. The sequence of transitions in a particular FSA corresponding to events in a portion of an execution of the system thus represents a flow from one state of the FSA to another.

To check a property of a concurrent system using INCA, an analyst specifies the ways that an execution might violate the property in terms of a sequence of intervals, or segments, of an execution. Consider a system in which event a can occur repeatedly and event b can occur at most once. Suppose that an analyst wants to show that an occurrence of event b can never be preceded by an occurrence of event *a* in any execution of the system. A violation of this property is an execution in which *a* occurs and then *b* occurs. In INCA this could be specified as a single interval running from the start of the execution until the occurrence of *b*, with the requirement that an *a* occur somewhere in the interval. (It could also be specified as a sequence of two intervals, the first running from the start of the execution until an occurrence of *a*, and the second starting immediately after the first and ending with *b*. The first type of specification is generally more efficient, but the second type may provide additional precision in some cases. This issue is discussed in more detail below.) INCA provides a query language allowing the analyst to specify various aspects of the intervals of execution. Standard INCA queries for a variety of common types of requirements are given at the Specification Patterns web site [2, 3].

By generating the equations describing flow within each FSA (requiring that the flow into a node equal the flow out) according to the specified sequence of intervals of a system execution, and adding equations and inequalities relating certain transitions in different FSAs according to the semantics of communication in the system, INCA produces a system of equations and inequalities. Any execution that satisfies the analyst's specification (and therefore violates the property being checked) corresponds to an integer solution of this system of equations and inequalities. INCA then uses standard integer linear programming (ILP) methods to determine whether there is an integer solution. If no integer solution exists, no execution can violate the property, and the property holds for all executions of the concurrent system. If there is an integer solution, however, we do not know that the property can be violated. The system of equations and inequalities represents only *necessary* conditions for the existence of an execution violating the property, and it is possible for a solution to exist that does not correspond to a real execution.

To see more concretely how this works, consider the Ada program shown in Figure 1. This program describes three concurrent processes (tasks). Task `t1` starts with a rendezvous with task `t2` at the entry `c`. It then enters a loop. At the select statement, `t1` nondeterministically chooses to rendezvous with `t2` at entry `a` or with `t3` at entry `b`, if both are ready to communicate at the appropriate entries. If `t1` accepts a communication from `t2` at entry `a`, it then enters a loop in which it accepts rendezvous at entry `a` until it accepts one at entry `c`. If `t1` instead accepts a communication from `t3` at entry `b`, it then tries forever to repeatedly rendezvous with `t2` at entry `a`. Task `t2` begins by calling entry `c`, and then enters a loop in which it calls entry `a`. Task

```
package simple is
  task t1 is
    entry a;
    entry b;
    entry c;
  end t1;
end simple;
```

```
task t2 is
end t2;

task t3 is
end t3;
```

```
package body simple is
  task body t1 is
  begin
    accept c;
    loop
      select
        accept a;
        loop
          select
            accept a;
          or
            accept c;
            exit;
          end select;
        end loop;
      or
        accept b;
        loop
          accept a;
        end loop;
      end select;
    end loop;
  end t1;
end simple;
```

```
task body t2 is
begin
  t1.c;
  loop
    t1.a;
  end loop;
end t2;
```

```
task body t3 is
begin
  t1.b;
end t3;
```

Fig. 1. A small example

t3 simply calls entry b once and then exits.

Figure 2 shows the FSAs constructed by INCA for this program. The states and transitions are numbered for reference. Each transition in this example represents the occurrence of a rendezvous between two tasks; in the figure, each transition is labeled with the entry at which the corresponding rendezvous takes place. (For this example, it is sufficient to label the transitions by the entry name. In practice, INCA identifies transitions representing rendezvous with the names of the calling and accepting tasks, the entry called, and the values of any parameters passed in the rendezvous. Note that we do not need to distinguish here between the "call" and the "accept"; we view the transition as representing the actual rendezvous involving both tasks. As will be seen below, INCA ensures the number of occurrences of transitions representing a given rendezvous is the same in the two tasks participating in that rendezvous.)

Suppose that we wish to check that an occurrence of a rendezvous at entry b cannot be pre-

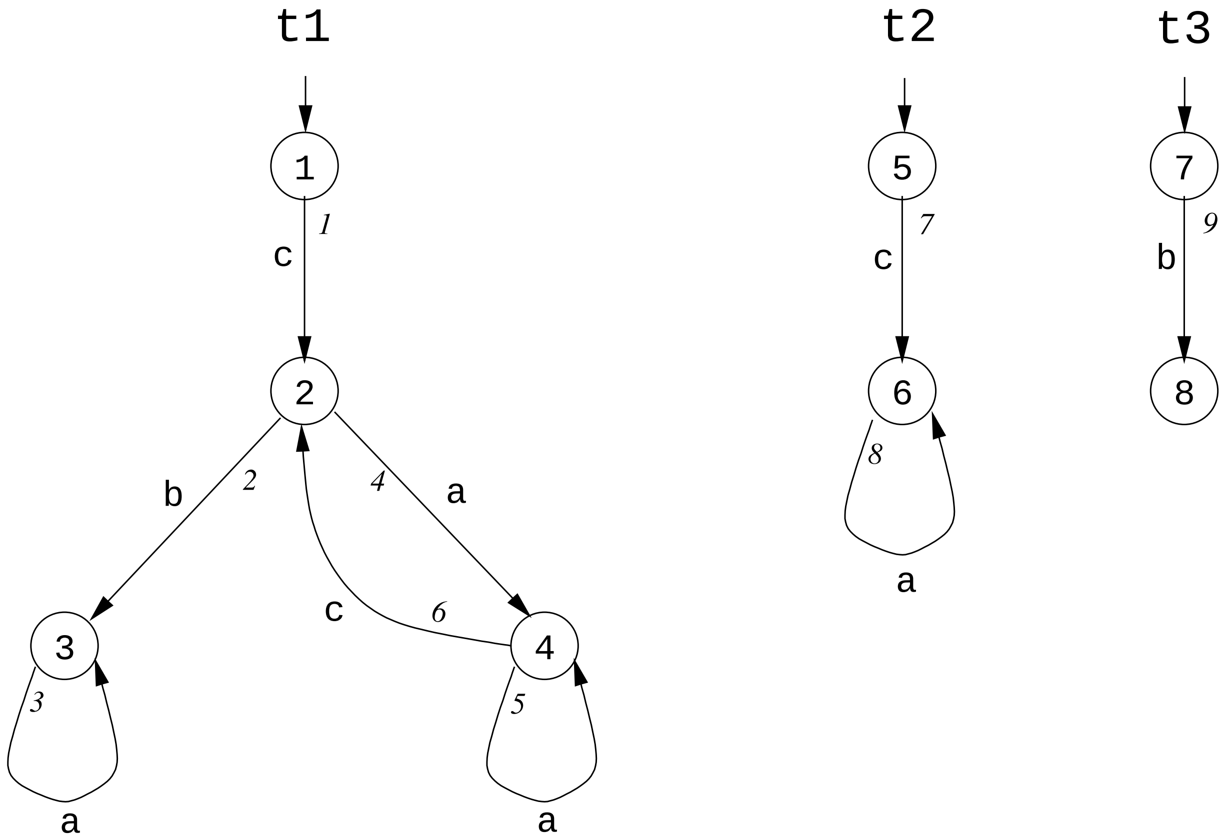

Fig. 2. FSAs for example

ceded by a rendezvous at entry `a`. As described earlier, we may specify the violation as an interval of an execution running from the start of execution until the occurrence of a rendezvous at `b` and containing a rendezvous at `a`. The flow equations for each task will then describe the possible flows from the initial state of the task to one of the states in which that task could be at the end of the interval.

Since the interval ends with a rendezvous at entry `b`, represented by the transition numbered 2 in the FSA corresponding to task `t1` and the transition numbered 9 in the FSA corresponding to task `t3`, we know that the FSA `t1` must be in state 3 and the FSA `t3` must be in state 8 at the end of the interval. Our flow equations for `t1` therefore describe flow starting in state 1 and ending in state 3, while the flow equations for `t3` describe flow starting in state 7 and ending in state 8. For `t2`, the fact that a rendezvous at `a` occurs in the interval implies that that FSA must be in state 6 at the end of the interval, so the flow equations for `t2` describe flow from state 5 to state 6.

To produce these flow equations, let x_i be a variable measuring the flow along the transition numbered i. At each state, we generate an equation setting the flow in equal to the flow out. We must, however, take into account the implicit flow of 1 into the initial state of each FSA and the implicit flow of 1 out of the end state of the flow. Thus, for example, the equation for state 1 is

$$1 = x_1 \tag{1}$$

since the flow in is 1 because state 1 is the initial state and the only flow out is on transition 1. Similarly, the equation for state 8 is

$$x_9 = 1 \tag{2}$$

since the only flow in is on transition 9 and there is implicit flow out of 1 since the flow in this FSA ends in state 8.

To complete the system of equations and inequalities, we must add equations to reflect the fact that the two tasks participating in a rendezvous must agree on the number of times it occurs. For instance, we need the equation

$$x_3 + x_4 + x_5 = x_8 \tag{3}$$

saying that the number of occurrences of the rendezvous at entry `a` in the FSA for `t1` is the same as in the FSA for `t2`. We also need an inequality to express the requirement that there is at least one occurrence of a rendezvous at `a`. We use

$$x_8 \geq 1 \tag{4}$$

to state this. The full system of equations and inequalities used to check the property that a rendezvous at entry `b` cannot be preceded by a rendezvous at entry `a` is shown in Figure 3. (The description here is actually somewhat oversimplified; INCA performs several optimizations to reduce the size of the system of inequalities and the real system of inequalities produced by INCA would be smaller. For example, INCA would observe that there cannot be flow along transition 3 in a violating execution (because the interval of execution must end with transition 2), and would eliminate the variable x_3 from the system. It would also do a form of constant propagation to eliminate other variables and equations.)

Essentially all research on finite-state verification tools can be viewed as aimed at ameliorating the state explosion problem for some interesting systems and properties. The approach taken by

Flow Equations:

State	Equation
1	$1 = x_1$
2	$x_1 + x_6 = x_2 + x_4$
3	$x_2 + x_3 = x_3 + 1$
4	$x_4 + x_5 = x_5 + x_6$
5	$1 = x_7$
6	$x_7 + x_8 = x_8 + 1$
7	$1 = x_9$
8	$x_9 = 1$

Communication Equations:

Entry	Equation
a	$x_3 + x_4 + x_5 = x_8$
b	$x_2 = x_9$
c	$x_1 + x_6 = x_7$

Requirement Inequality:

a occurs	$x_8 \geq 1$

Fig. 3. System of equations and inequalities for example

INCA avoids enumerating the reachable states of the system. The size of the system of equations and inequalities is essentially linear in the number of processes in the system (assuming the size of each process is bounded). Furthermore, the use of properly chosen cost functions in solving the problems can guide the search for a solution. ILP is itself an *NP*-hard problem in general, and the standard techniques for solving ILP problems (branch-and-bound methods) are potentially exponential. In practice, however, the ILP problems generated from concurrent systems have large totally unimodular subproblems and seem particularly easy to solve. Experience suggests that the time to solve these problems grows approximately quadratically with the size of the system of inequalities (and thus with the number of processes in the system).

Comparisons of this approach [4–7] with other finite-state verification methods show that the performance of each method varies considerably with the system and property being verified, but that INCA frequently performs as well as, or better than, such tools as S [8] and SMV [9]. The INCA approach has also been extended to check timing properties of real-time systems [10, 11] and to prove trace equivalence of certain classes of systems [12].

B. Sources of Imprecision

The systems of equations and inequalities generated by INCA represent necessary conditions for there to be a violation of the property being verified. As noted earlier, however, these conditions are not, in general, sufficient to guarantee that the property can actually be violated. A solution of the system of equations and inequalities may not correspond to a real execution.

There are two main reasons for this. The first has to do with the order in which events occur. Strictly speaking, the equations and inequalities generated by INCA refer only to the total number of occurrences of the various events in each interval of the execution, and do not directly impose restrictions on the order in which those events occur within the interval. In fact, the flow equations for a single FSA typically imply fairly strong conditions on order, but the communication equations relating the occurrence of events in different FSAs do not impose strong restrictions on the order of occurrence of events from different processes. To see why, consider a system comprising two processes. The first process begins by trying to communicate with the second process on channel A and then, after completing that communication, tries to communicate with the second process on channel B. The second process tries to complete the communications in the reverse order. This system will obviously deadlock, but the equations generated by INCA would say only that the number of communications on each channel in the first process is equal to the number in the second process, allowing a solution in which each communication occurs. (This is again a slight over-simplification. INCA would actually detect the deadlock in this case, but not in more complicated examples with several processes.) The only mechanism INCA provides for directly constraining the order of events in different processes is the use of additional intervals of the execution. While this is often enough to eliminate solutions that do not correspond to real executions of the system, it is expensive and restricts the range of application of INCA. We will return to this point later.

The second source of imprecision is the existence of cycles in the FSAs. Consider the flow equation for state 3 that is shown in Figure 3. Transition 3 is a self-loop at state 3, and flow along that transition counts both as flow into state 3 and out of state 3. The equation $x_2 + x_3 = x_3 + 1$ does not constrain the variable x_3 at all; we can simply cancel the x_3 terms. Similarly, the variables x_5 and x_8 are not constrained by the flow equations in which they appear. These variables are constrained only by the communication equation that says $x_2 + x_3 + x_5 = x_8$. Since three of these variables are otherwise unconstrained, this equation does not restrict the solution set.

In fact, although the system of Figure 1 has no execution in which a prefix ending with a rendezvous at entry `b` contains a rendezvous at entry `a`, there is a solution to the system of equations and inequalities shown in Figure 3 with x_1, x_2, x_5, x_7, x_8, and x_9 all equal to 1, and x_3, x_4, and x_6 all equal to 0. In this solution, the requirement that the number of rendezvous at `a` be at least 1 is met by setting the unconstrained variables x_5 and x_8 to 1. Figure 4 shows the FSAs with the transitions having flow indicated by bold arcs. The flow in the FSA for `t1` has two connected components, one from the initial state to state 3, as expected, and one made up of flow in the cycle at state 4, not connected to the flow from state 1 to state 3. It is obvious that the flow in each FSA corresponding to an actual execution must be connected, so this is a spurious solution, one that does not correspond to a real execution.

This example illustrates the problem but is not of much independent interest. The same problem, however, occurs with some frequency in the analysis of more interesting systems. For instance, in our recent analysis [4] of the Chiron user interface development system, we encountered solutions with disconnected cycles in trying to verify 2 of the 10 properties we checked. In those cases, we were able to verify the high-level requirements by reformulating the properties being checked. We subdivided some intervals to force events in different parts of the spurious cycles to occur in different intervals, verified other properties that allowed us to eliminate some solutions, or chose other events to represent the high-level requirement. These modifications, however, represent a considerable expense in increased analyst effort and verification time, and made the properties being checked harder to understand and validate in terms of the high-level requirements. In the next section, we describe a technique for eliminating these solutions with more than one component of flow in an FSA.

III. E S C

A. A Straightforward Approach

A related problem is well known in the optimization literature. When formulating the Traveling Salesman Problem as an integer programming problem, it is essential to ensure that the

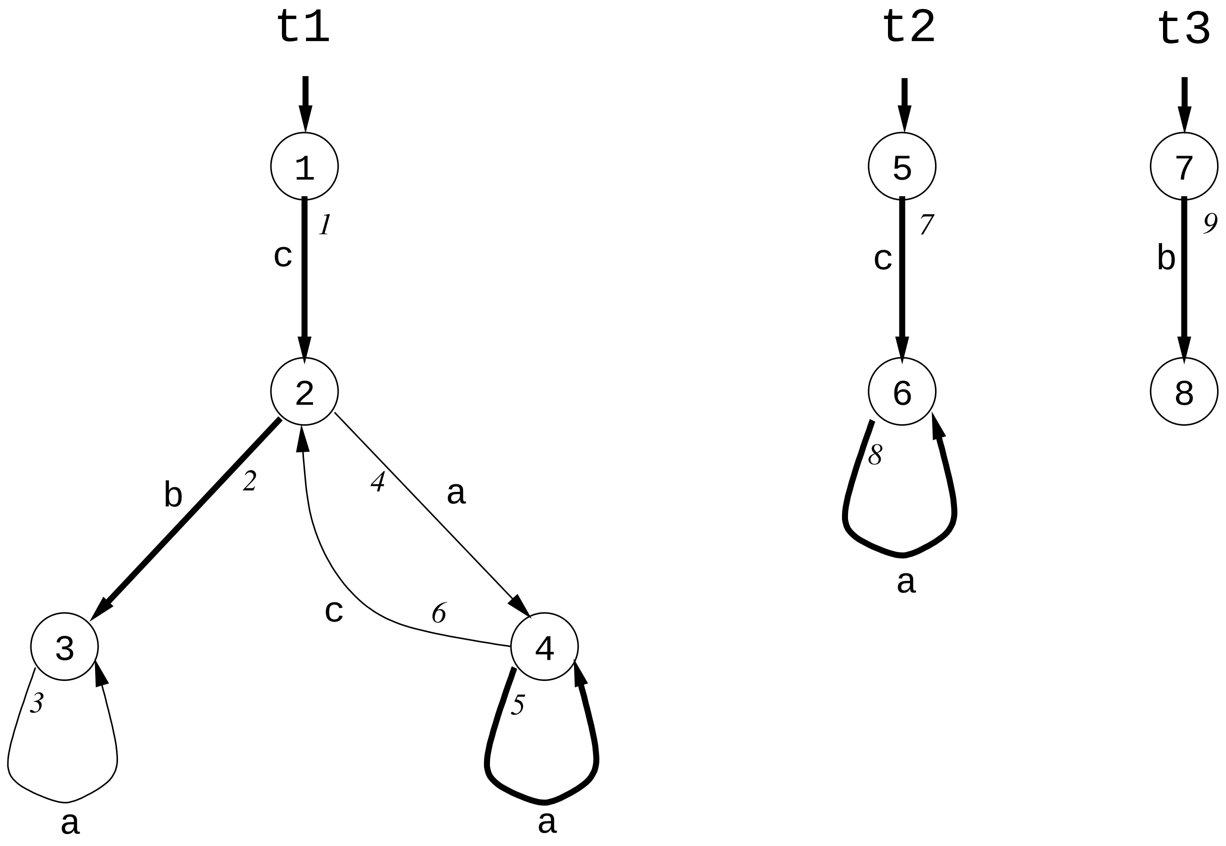

Fig. 4. Solution with disconnected cycle

solution represents a single tour visiting all the cities, rather than a collection of disconnected subtours each visiting a proper subset of the cities. A standard approach for eliminating solutions with disconnected subtours is to add inequalities that prevent the solution from visiting cities in a subset U unless the solution includes an arc from a city not in U to one in U. Thus, if the variable $x_{i,j}$ is 1 if the solution represents a tour in which the salesman goes directly from city i to city j, and 0 otherwise, the standard formulation of the Traveling Salesman problem would include, for each j, the inequality

$$\sum_{i} x_{i,j} = 1 \tag{5}$$

to enforce the requirement that each city is entered and left exactly once. To eliminate the possibility of a subtour in the subset U we would add the inequality

$$\sum_{i \notin U, j \in U} x_{i,j} \geq 1, \tag{6}$$

which requires that the salesman travel from a city outside U to a city in U. (Of course, we need an inequality like (6) for every subset U of size at least 2 and at most $N - 2$, where N is the number of cities.)

In our case, to prevent a solution in which there is flow in a disconnected cycle C, we can add an inequality requiring that, when there is flow in C, there must be flow entering C from outside. This is a little more complicated than the situation for the Traveling Salesman Problem. In that case, we know by (5) that the the solution must enter each city exactly once. In our case, we do not want to require flow into one of the states making up C *unless* there is flow along one of the transitions in C. For instance, we only want to require flow on transition 4 in our example when there is flow on transition 5. To do this in general, we would need a quadratic inequality such as

$$x_4 x_5 \geq x_5. \tag{7}$$

Integer quadratic programming is, however, much harder than integer linear programming and we would like to avoid introducing quadratic inequalities. The standard technique is to impose an upper bound B on all the variables (i.e., to assume that no transition occurs more than B times), and to replace the quadratic inequality (7) with the linear inequality

$$x_5 - Bx_4 \leq 0. \tag{8}$$

The integer solutions of (7) having $x_4, x_5 \leq B$ are exactly the same as those of (8). (We note that imposing an upper bound on all the variables would mean that INCA's analysis is no longer strictly conservative. If the system of inequalities has no solutions with the x_i all less than or equal to B, we only know that no execution on which each transition occurs at most B times can violate the property. Since B can be taken to be quite large, such as 10,000 or 100,000, this restriction is unlikely to be a serious one in practice.)

The problem with these approaches is that they may require too many extra inequalities. The number of subtours that have to be eliminated in the Traveling Salesman Problem is essentially the number of subsets of the set of cities and is clearly exponential in the number of cities. Similarly, the number of cycles in an FSA can be essentially equal to the number of subsets of its set of states. We have constructed a small concurrent Ada program with only 90 lines of code in which the FSA for one task has only 42 states but has 1,160,290,624 distinct subsets of states each forming at least one cycle. An integer programming problem with that many inequalities is completely intractable. A better method is required.

B. A More Practical Method

In this section, we describe a method for preventing spurious cycles for which the number of additional variables and inequalities is linear in the size of the program being analyzed.

The basic idea is as follows. Suppose we have a solution to the system of equations and inequalities originally generated by INCA. For each FSA, the solution determines a subgraph G' consisting of the edges with positive flow and the vertices with flow in or out. If G' is not connected, i.e., if some vertex is not reachable from the initial vertex, the solution must involve a spurious cycle in that FSA. To show that G' is connected, it is sufficient to construct a subgraph of G' having the same vertex set as G' and such that *(i)* if there is flow along any edge into a vertex v in the solution, some edge terminating in v and having positive flow in the solution must occur in the subgraph, and *(ii)* each vertex v of the subgraph can be assigned a "depth" d_v in such a way that the depth of a given vertex is greater than that of any of its predecessors in the subgraph. The second condition makes the subgraph acyclic, and then the first condition ensures that each vertex with incoming flow in the solution has an incoming edge in the subgraph and is therefore reachable from the initial vertex.

If the original solution has no disconnected cycles, we can choose for our subgraph a spanning tree for the edges with flow and take the depth of a vertex to be the distance from the root of the tree to the vertex. If the solution has a disconnected cycle C, however, we cannot construct such a subgraph. To see why, suppose we could construct the subgraph, and let v be a vertex in C for which $d_v \leq d_u$ for all $u \in C$. Since there is flow into v in the solution, v must have some predecessor u in the subgraph. Since the cycle C is disconnected from the flow starting at the initial state of the FSA, the state u must also lie in C. But if u is a predecessor of v in the subgraph, we have $d_v > d_u$, contradicting the minimality of d_v on C.

Of course, we do not want to consider the possible solutions to the system of equations and inequalities generated by INCA one at a time, attempting to construct the subgraph separately for each solution. Instead, we add new variables and inequalities, leading to an augmented system of equations and inequalities whose integer solutions correspond exactly to the integer solutions of the original system for which the appropriate subgraph can be constructed.

1) The Flowgraph: In general, a query can specify more than one interval, so the situation is slightly more complicated than that illustrated in our example. In the general case, INCA constructs a *flowgraph* as follows. First, it creates one copy of each FSA for each interval specified in the query. The FSAs for each interval can then be optimized independently, removing unnecessary states or transitions based on the restrictions imposed for that interval in the query. As discussed in Section II-A, INCA can analyze the query to determine the possible states in which each FSA could be at the end of the interval. Given such a state in an FSA for a task in an interval (other than the last one), INCA adds a "connect" edge to the corresponding state in the FSA for that task in the next interval. These edges, which do not correspond to events in the execution of the system, allow flow to pass from one interval to the next. INCA adds an initial vertex, v_I, with connect edges to the initial states of the FSAs in the first interval, and a final vertex, v_F with incoming connect edges from each of the possible end states of FSAs in the final interval. This flowgraph is the structure that INCA actually uses to generate the system of equations and inequalities. Note that several different edges in this graph may correspond to a single edge in an FSA, representing flow along that edge in different intervals.

Figure 5 shows the flowgraph generated from the system of Figure 2 for a query with two intervals that describes executions on which a rendezvous at `a` occurs before the rendezvous at `b` does. (This is the two-interval version of the query used to check the property that a `b` can never be preceded by an `a`, as described in Section II-A.) The query specifies that the first interval ends as soon as an `a` occurs and contains no `b`. The second interval ends as soon as the `b` occurs. The connect edges are shown with dashed lines. The vertices (other than the initial and final ones) and edges are labeled the same way as the corresponding vertices and edges in Figure 2. Since the first interval ends with the first `a` and does not allow a `b`, INCA can determine that the last event in task `t1` in the first interval must be the transition on edge 4 from state 2 to state 4, and the last event in task `t2` must be the transition on edge 8 from state 6 to itself. Because no `b` is allowed in the first interval, INCA can prune edges 2 and 3 from the FSA for `t1` and edge 9 from the FSA for `t3` in the first interval.

INCA associates a variable x_e to each edge e, and generates flow equations as follows. For each vertex other than the initial or final vertices, INCA generates the equation *flow-in* = *flow-out*, where *flow-in* is the sum of the variables associated with edges entering the vertex

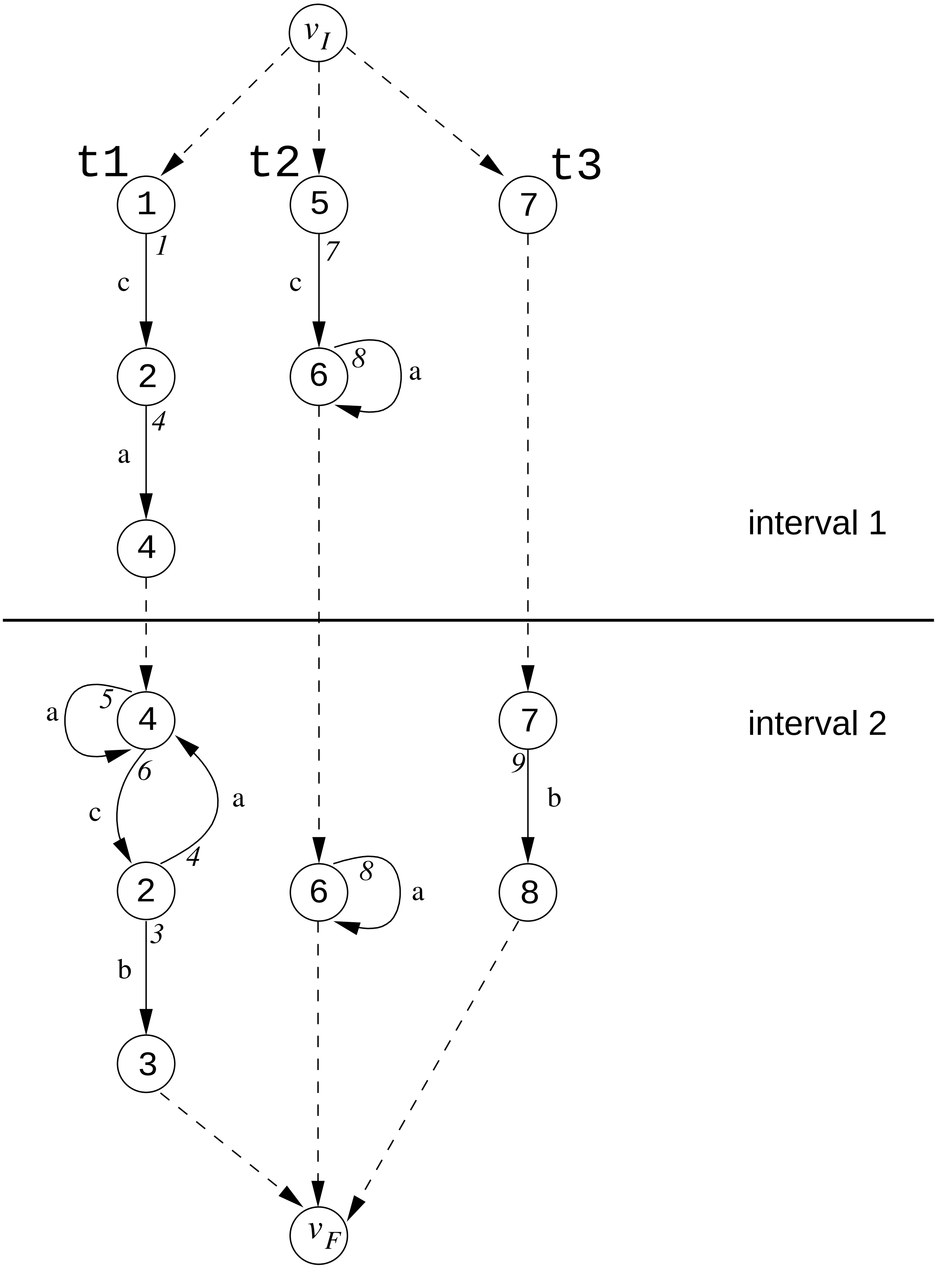

Fig. 5. Flowgraph for example with two-interval query

and *flow-out* is the sum of the variables associated with edges leaving the vertex. For each task, INCA generates an equation setting the flow from the initial node to the start node of that task equal to one, and an equation setting the sum of the flow along edges from nodes of that task to the final node equal to one. Communication equations of the type described in Section II-A are generated for each interval to ensure that the communicating tasks agree on the number of communications that occur in each interval, and additional constraints are added to reflect additional requirements or restrictions imposed by the query on the possible events occurring in the different intervals. (The fact that the different edges in the flowgraph, representing flow in different intervals, correspond to the same transition in an FSA representing a task makes it possible to require an event to occur in one interval and forbid it in another.)

2) The Augmented System of Inequalities: We now describe precisely the procedure for generating the augmented system of equations and inequalities that eliminates solutions with disconnected cycles.

We will say that an *ILP problem* $\mathcal{P}$ is a set of integer variables with upper and lower bounds specified for each variable, together with a set of linear equations and inequalities in those variables such that the all the coefficients are integers. (The bounds on a variable may be taken to be infinite.) A *solution* to an ILP problem $\mathcal{P}$ is an assignment of integers to the variables such that the value of each variable lies between its upper and lower bounds and all the equations and inequalities are satisfied. (In standard usage, an ILP problem would also include a linear function of the variables and the task is to find a solution that maximizes or minimizes this objective function. In our case, we are primarily interested only in the *feasibility* of the ILP problem, that is, whether or not there are any solutions, and in this paper we can ignore the objective function. In applying INCA, we use the objective function to improve performance when there are solutions to the ILP problem.)

Let G be a directed graph with a specified initial vertex v_I and a specified final vertex v_F such that v_I has no incoming edges and v_F has no outgoing edges. Let $\mathcal{P}$ be an ILP problem containing *(i)* a variable x_e with lower bound of 0 for each edge e in G, and *(ii)* the flow equation *flow-in* = *flow-out* for each vertex in G other than v_I and v_F. ($\mathcal{P}$ may contain additional variables and constraints, we are just requiring that it contain at least these.)

Given a solution to $\mathcal{P}$, we say that an edge *e has flow* if $x_e > 0$, and we say that a vertex *v has flow* if some edge entering or leaving v has flow. By the *flow subgraph of G corresponding to* *the solution*, we mean the subgraph G' of G consisting of all the vertices and edges with flow. We say that the solution is *connected with respect to* G if G' is connected, that is, if every vertex in G' is reachable from the initial vertex v_I. We may suppress the qualification "with respect to G" if the graph G is clear from context.

The idea, as described at the beginning of Section III-B, is to construct a subgraph of G' having the same set of vertices as G', but possibly fewer edges. We require that *(i)* for each vertex $v \neq v_I$ of G', some edge of G' entering v must be in the subgraph; and *(ii)* each vertex v can be assigned a "depth" d_v in such a way that the depth of a given vertex is greater than the depth of any of its predecessors in the subgraph. Our goal is to describe an augmented ILP problem $\mathcal{P}'$ such that a solution of $\mathcal{P}$ can be extended to a solution of $\mathcal{P}'$ if and only if its flow subgraph G' has a subgraph satisfying conditions *(i)* and *(ii)*.

For each edge e in G, we introduce a new variable s_e with bounds

$$0 \leq s_e \leq 1. \tag{9}$$

(Note that the imposition of an upper or lower bound on a variable can of course be thought of as adding an inequality, but is usually handled somewhat differently by ILP packages. In this discussion, we will separate the imposition of bounds from the introduction of new inequalities.) This variable will be 1 if the corresponding edge is in the subgraph, and 0 otherwise.

For each vertex v in G, we introduce a new variable d_v with bounds

$$0 \leq d_v \leq N, \tag{10}$$

where N is some integer which is at least the maximum length of any non-self-intersecting path through the graph. For instance, N can be taken to be the number of vertices in G. The variable d_v will be the depth of v.

We then generate inequalities involving these new variables. For each edge $e\colon u \to v$, we generate the inequalities

$$x_e \geq s_e \tag{11}$$

$$d_v \geq d_u + (N+1)s_e - N. \tag{12}$$

The first inequality says that s_e must be 0 if x_e is 0, so that the corresponding edge can be in the subgraph only if the solution has positive flow along that edge. The second inequality requires that d_v be greater than d_u if the edge from u to v is in the subgraph. If the edge is not in the subgraph (i.e., if s_e is 0), the inequality reads $d_v \geq d_u - N$, and the bounds on d_v and d_u make that vacuous.

Finally, let B be a fixed positive integer, and impose the upper bound $x_e \leq B$ for each e. (As noted earlier, B can be taken to be quite large.) For each vertex v of G, other than the initial vertex, we generate the inequality

$$B|\mathrm{In}(v)| \sum_{e \in \mathrm{In}(v)} s_e \geq \sum_{e \in \mathrm{In}(v)} x_e, \tag{13}$$

where $\mathrm{In}(v)$ denotes the set of edges entering v. By imposing the upper bound of B on the x_e, we see that (13) will hold if $x_e = 0$ for all $e \in \mathrm{In}(v)$ or if $s_e = 1$ for at least one such e. This is how we enforce the requirement that each vertex with flow has some incoming edge in the subgraph of G'.

We have added $V + E$ new bounded variables and $V + 2E - 1$ new constraints, where V is the number of vertices in the graph and E is the number of edges. Let $\mathcal{P}'$ be the ILP problem obtained from $\mathcal{P}$ and G by adding the variables and inequalities in this fashion. We have the following theorem.

Theorem. *Let G, $\mathcal{P}$, and $\mathcal{P}'$ be as above. A solution of $\mathcal{P}'$ assigns values to all the variables in $\mathcal{P}$ as well as additional variables; we thus obtain a solution to $\mathcal{P}$ from a solution to $\mathcal{P}'$ by projection. The set of connected solutions of $\mathcal{P}$ with each $x_e \leq B$ is exactly equal to the set of projections of solutions of $\mathcal{P}'$.*

Proof: There are two things we must show. First, we must show that, given any connected solution to $\mathcal{P}$ with all $x_e \leq B$, there are values that can be assigned to the new variables to give a solution to $\mathcal{P}'$. Second, we must prove that the projection of any solution for $\mathcal{P}'$ is a connected solution for $\mathcal{P}$. We tackle these in order.

Suppose we are given a connected solution for $\mathcal{P}$. Then in the flow-subgraph G', every vertex is reachable from the initial vertex v_I. So G' has a spanning tree T rooted at v_I, i.e., T is a subgraph of G' that is a tree with root v_I and that contains all the vertices of G'. For each edge e in G, let

$$s_e = \begin{cases} 1 & \text{if } e \text{ is in } T \\ 0 & \text{otherwise.} \end{cases}$$

For each vertex v in G', let d_v be the depth of v in T. For vertices v not in G', d_v may be assigned any value between its bounds.

We claim this is a solution for $\mathcal{P}'$. Indeed, inequality (11) follows from the fact that $T \subseteq G'$. To see that (12) holds, suppose we are given an edge $e\colon u \to v$. If e is in T, then $d_v = d_u + 1$, and (12) reduces to the statement $d_v \geq d_u + 1$, which certainly holds. On the other hand, if e is not in T, then (12) becomes $d_v \geq d_u - N$, which must hold because $d_v \geq 0$ and $d_u \leq N$. Now consider a vertex $v \neq v_I$ in G. If v does not have flow then (13) holds trivially as its right hand side is 0. If v does have flow, then it is in T, and therefore $s_e = 1$ for some edge e occurring in the sum. So the left hand side of (13) is at least $|\mathrm{In}(v)|B$. On the other hand, there are $|\mathrm{In}(v)|$ terms in the sum on the right hand side and each $x_e \leq B$, so

$$\sum_e x_e \leq |\mathrm{In}(v)|B,$$

and (13) holds in this case as well.

Now suppose we are given any solution to $\mathcal{P}'$. We wish to show that the projection of this solution is connected. Let G' be the flow subgraph for the projected solution. Let U be the set of vertices in G' that are not reachable in G' from the initial vertex. To say that the solution is connected is equivalent to saying that U is empty. So suppose U is not empty, and let v be a vertex in U for which $d_v \leq d_w$ for all w in U. Since v has flow and v is not the initial vertex, v has incoming flow (either because $v = v_F$ or by the flow equation for v). This implies that the right hand side of (13) is at least 1, so for some edge $e\colon u \to v$ we have $s_e = 1$. Now (11) implies $x_e \geq 1$. Since u has flow out to v, if u were reachable from v_I, v would also be reachable. But $v \in U$ is not reachable, so we must have $u \in U$ as well. Now (12) implies

$$d_v \geq d_u + 1,$$

which contradicts the minimality of d_v. ■

3) Local Application of Cycle Elimination: For systems generated by INCA, it is often not necessary to apply the cycle elimination algorithm over the entire flowgraph. One reason for this is the following. As there are no edges from vertices of one interval to vertices of a previous interval, any cycle must be contained in a single interval of the flowgraph. Furthermore, as there are no edges from vertices of one task to vertices of another task, a cycle must be contained within a single task of a single interval. Experience has shown us that, as far as cycles are concerned, these task-interval "sectors" of the flowgraph often behave independently: existence of spurious cycles in one sector usually has no bearing on the existence of spurious cycles in another sector.

Although some of the preliminary experiments described below suggest that the expense of applying our cycle elimination algorithm routinely may not be excessive, we expect that it will often be applied only when an attempt to verify a property without using cycle elimination has produced a solution with spurious cycles. So, once a spurious cycle has been encountered this way, it would be useful if there were a way to generate cycle elimination constraints for only the relevant sector of the flowgraph, and therefore save on the number of new constraints and variables generated—and probably analysis time and memory as well.

In fact this requires only a slight modification of the algorithm. Suppose we are given a graph G and an ILP problem $\mathcal{P}$ as in Section III-B.2. Let V' be a subset of the vertices of G not containing the initial or final vertices. We say that a solution to $\mathcal{P}$ is *V'-connected with respect to G* if, in the flow subgraph G' corresponding to that solution, every vertex in V' is reachable from some vertex outside V'. (A solution has a spurious cycle entirely contained in V' if and only if it is not V'-connected.) Suppose that we want to eliminate solutions to $\mathcal{P}$ that are not V'-connected. We construct a new graph from the vertices in V' and the edges of G entering or leaving those vertices, and apply the augmentation algorithm described above to that graph. This produces a set of new variables and constraints. Adding those to $\mathcal{P}$ and imposing the upper bound of B on the x_e, we get an augmented system whose solutions project to the solutions of $\mathcal{P}$ having no disconnected cycles contained in V'.

Let G_1 be the subgraph of G obtained by first deleting all edges that do not enter or leave a vertex in V', and then deleting all vertices with no incoming or outgoing edges. The subgraph G_1 contains all the vertices in V' and all the vertices with an edge to or from a vertex in V', and all the edges of G that enter or leave vertices in V'.

We add a new initial vertex w_I and a final vertex w_F to G_1. For any edge $e\colon u \to v$ in G_1 from a vertex u not in V' to a vertex v in V', we replace e by a new edge $e'\colon w_I \to v$. For any edge $f\colon v \to w$ from a vertex v in V' to a vertex w not in V', we replace f by a new edge $f'\colon v \to w_F$. We then remove all vertices other than w_I, w_F, and those in V'. Let $\hat{G}$ be this new graph. (Another way to think of this process is the following. Let U be the set of vertices u of G_1 that do not belong to V' but have an outgoing edge to a vertex in V'. Let W be the set of vertices of G_1 that do not belong to V' but have an incoming edge from a vertex in V'. The graph $\hat{G}$ is obtained from G_1 by collapsing each of the sets U and W to a single vertex.)

Corollary. *Given G and $\mathcal{P}$ as in Section III-B.2, and a subset V' of the vertices of V not containing v_I or v_F, define $\hat{G}$ as above. Let $\hat{\mathcal{P}}$ be the augmented system obtained by applying the algorithm of Section III-B.2 to $\hat{G}$ and $\mathcal{P}$. Then the set of solutions of $\mathcal{P}$ that are V'-connected and have all $x_e \leq B$ is exactly equal to the set of projections of solutions of $\hat{\mathcal{P}}$.*

Proof: The graph $\hat{G}$ and the system $\mathcal{P}$ satisfy the hypotheses of the Theorem of Section III-B.2. Moreover, the solutions of $\mathcal{P}$ that are connected with respect to $\hat{G}$ are exactly the solutions that are V'-connected with respect to G. ■

IV. P E

The current version of INCA consists of about 12,000 lines of Common Lisp. INCA writes out a file describing the ILP problem in a standard format (the MPS format), and we use a commercial package called CPLEX to read this file and solve the system. (We also use a separate program to translate Ada programs into the native input language of INCA). The optimizations INCA uses to reduce the number of variables and inequalities make the introduction of new variables and inequalities somewhat complicated, and integrating our method into INCA will involve a substantial programming effort. For our initial exploration of the effect of applying our method, we have therefore chosen to proceed by modifying the MPS file produced by INCA. We have written a Java program that reads this file, and another file describing the flowgraph, and produces a new MPS file representing the augmented system of equations and inequalities. We can then compare the performance of CPLEX on the original system and the augmented system. We are not as interested in the time it takes to run the Java program because the algorithm which this program implements is very simple and is clearly linear in the number of nodes and edges in the graph used for cycle elimination. The algorithms used for solving ILP problems, on the other hand, are extremely complex and we have no theoretical way of estimating the time it takes to solve the ILP systems we produce. If there is a practical barrier to our cycle elimination technique, it will arise from solving the ILP systems, not from generating them.

In any case, the times for the Java program ranged from 1 to 22 seconds for the example in Section A, 2 to 14 seconds for Section B, 2 to 28 seconds for Section C, and 1 to 2 seconds for

Section D. The CPLEX times will be given in more detail below.

For these experiments, we used INCA version 3.4, Harlequin Lispworks 4.1.0, Java 2 SDK 1.3.0, and CPLEX version 7.0 on a Sun Enterprise 3500 with two processors and 2 GB of memory, running Solaris 2.8. The upper bound B representing the maximum number of times an edge may be traversed in a violating execution was taken to be 10,000. We used the default options on CPLEX, except for the following changes: MIP EMPHASIS was set to 1, MIP LIMITS TREEMEMORY to 2000, and MIP LIMITS SOLUTIONS to 1. (The first option affects choices made in the branch-and-bound algorithm, the second controls the storage of branch-and-bound nodes, and the third stops the search as soon as an integer solution is found.) For each ILP problem, we ran CPLEX five times and took the average time. The times reported here were collected using the `timex` command, and include both user and system time.

A. A Scalable Version of the Example from Section 2

For the first experiment, we created a scalable version of the simple example described in Section II-A. Given an integer $n \geq 2$, we modified the Ada program in Figure 1 to have n copies of task `t2` and to have $n + 1$ alternatives in the select statement. Each of the new copies of task `t2` calls the same entries in `t1`. (In detail, we replaced task `t2` with n copies of itself, calling these `tc1`,...,`tc`n. In the body of `t1`, we replaced the first `accept c` line with n copies of itself and replaced the body of text beginning with the first `accept a` and ending with the last `or` with n copies of itself.)

As before, we wish to verify that one cannot have the rendezvous at entry `b` preceded by a rendezvous at entry `a`. Using the standard 1-interval query, for each n, INCA finds a spurious solution involving a disconnected cycle in `t1`. After applying our cycle elimination algorithm to the sector of the flowgraph involving `t1`, we get an ILP problem that CPLEX reports has no integer solutions, thus verifying the property.

There is another way to get around the cycle problem in this case. As we mentioned in Section II-A, one can express the query using two intervals: the first interval begins with the start of execution, ends with an `a`, and does not contain a `b`, and the second interval ends with a `b`. Because of the trimming that INCA performs on each interval of the flowgraph, the opportunity for a spurious cycle is removed. So using this 2-interval version of the query, we were also able to verify the property.

At this point we are considering three distinct families of ILP systems:

- $\mathcal{P}_1(n)$: the system produced by INCA for the single-interval query (which has a spurious solution, so the analysis is inconclusive),
- $\mathcal{P}_2(n)$: the system produced by INCA for the two-interval query (which is inconsistent, so the property is verified), and
- $\mathcal{P}_3(n)$: the system obtained by applying cycle elimination to $\mathcal{P}_1(n)$ (which is also inconsistent and verifies the property).

For $n \geq 3$, the number of variables in $\mathcal{P}_1(n)$ is $4n^2 + 2n$, and the number of constraints (equations and inequalities) is $4n + 2$. The graph $\hat{G}$ used for cycle elimination has $2n + 4$ vertices and $4n^2 + 3$ edges; hence the algorithm adds $4n^2 + 2n + 7$ variables and $8n^2 + 2n + 9$ constraints to produce $\mathcal{P}_3(n)$.

The number of variables in system $\mathcal{P}_2(n)$ is $5n^2 + 7n$ and the number of constraints is $11n + 3$ (for $n \geq 2$). Hence in size $\mathcal{P}_2(n)$ falls between $\mathcal{P}_1(n)$ and $\mathcal{P}_3(n)$.

In Figure 6, we show the time it takes CPLEX to analyze each of these systems, for $n = 1, \ldots, 40$. All of these times are very modest—under 15 seconds—and are in fact dwarfed by the time it takes INCA to generate either ILP system. It is also clear that for this problem, as far as CPLEX time is concerned, using the 2-interval query is better than the single-interval query plus cycle elimination. However, it took INCA approximately 3 hours to generate $\mathcal{P}_1(40)$, and it took the Java program 23 seconds to apply the cycle elimination algorithm to produce $\mathcal{P}_3(40)$, whereas it took INCA approximately 10 hours to generate $\mathcal{P}_2(40)$. So when total analysis time is taken into consideration the cycle elimination technique wins hands down. Nevertheless, it does seem that for large n, the substantial increase in the number of constraints in $\mathcal{P}_3(n)$ due to the large number of edges in the FSA for `t1`, begins to have a significant impact on the time to solve the ILP problem.

B. Spurious Cycles in Chiron

The second experiment involves the Chiron user interface system [13]. A Chiron client comprises some abstract data types to be depicted, *artists* that maintain mappings between these ADTs and the visual objects appearing on the screen, and runtime components that provide coordination. In particular, certain *events* indicating changes in the state of the ADTs are defined, and an `ADT_Wrapper` task notifies a `Dispatcher` task whenever an event occurs. The

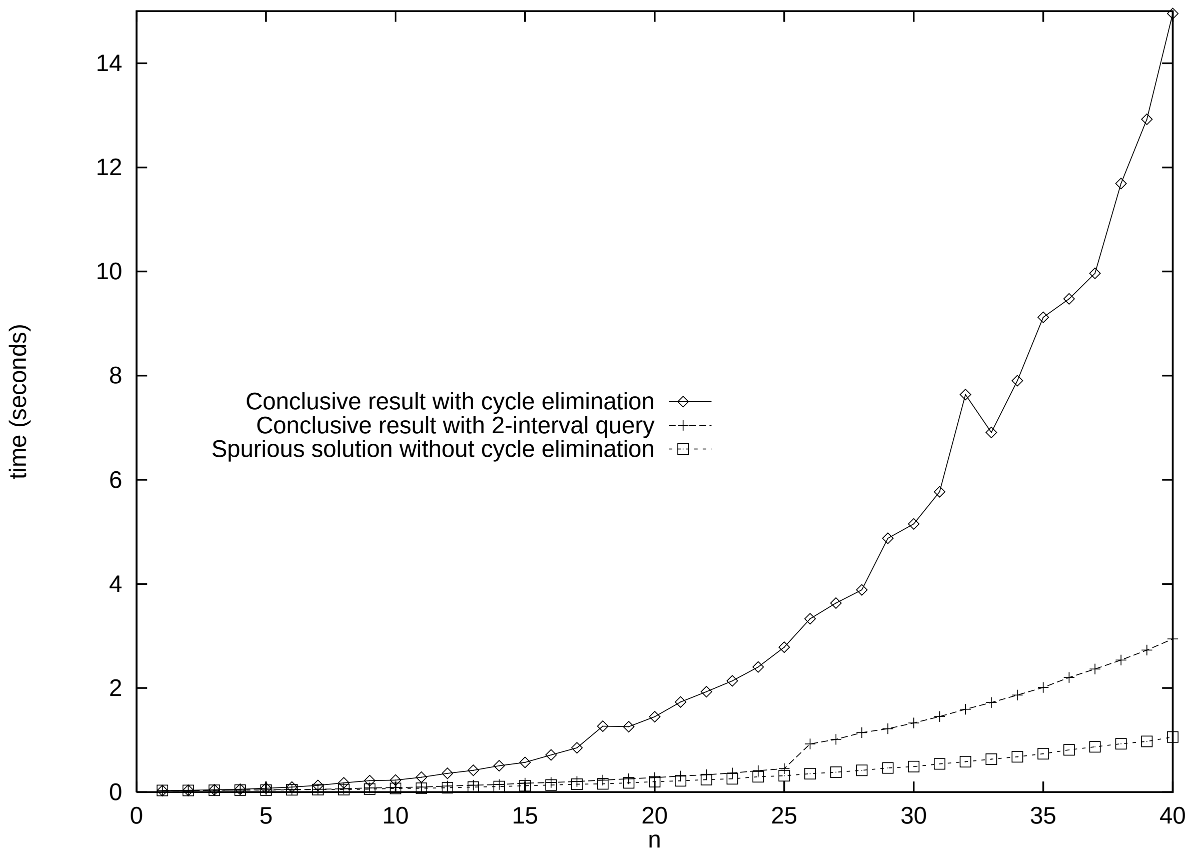

Fig. 6. CPLEX times for scaled simple example

`Dispatcher` maintains an array for each event that records which artists are interested in being notified of that event. (Artists register and unregister for an event to indicate their current interest in being notified.) After receiving the event from the `ADT_Wrapper`, the `Dispatcher` then loops through the artists in the appropriate array and calls an entry in each artist to notify it of the event. The Chiron architecture is highly concurrent and even a toy Chiron interface represents about 1000 lines of Ada code. In [4], we compared the performance of several finite-state verification tools (FLAVERS [14], INCA, SMV, and S) in checking a number of properties of a Chiron interface with two artists and *n* different kinds of events, for *n* ranging from 2 to 70.

One of the properties we wish to verify about this system, called Property 4 in [4], is that the `Dispatcher` notifies the artists of the *right* event. For example, if the `Dispatcher` receives event `e1` from the `ADT_Wrapper`, we wish to show that it does not notify any artist of some other event instead. To formulate this property as an INCA query takes 2 intervals.

We were in fact able to verify this property using INCA, but only in systems where the number of kinds of events, n, is at most 5. (FLAVERS and S were able to verify this property up to at least $n = 40$ and $n = 36$, respectively.) To scale the problem further with INCA, we needed to decompose the `Dispatcher` task into a subsystem. This entails creating a new task `Dispatch_ei`, for $i = 1, \ldots, n$, which maintains the array for event `ei`. The `Dispatcher` task itself is left as an interface which just passes register, unregister, and notification requests on to the appropriate `Dispatch_ei` in a way such that no additional concurrency is introduced. (If the internal communications of the Dispatcher subsystem are hidden, the new system is observationally equivalent to the original one.) This decomposed system has the advantage that as n increases, the size of each `Dispatch_ei` FSA remains constant, although the number of these tasks increases. In general this decomposition greatly improves the performance of INCA. For example, we were now able to verify several of the other properties in sizes up to $n = 70$. But attempting to verify property 4 with the decomposed Dispatcher task gave an inconclusive result. The problem is a disconnected cycle in the task `Dispatch_e1` in the second interval.

In [4], we got around this problem by reformulating the property using different events to represent the high-level requirement. This depended on the prior verification of other properties relating the events used in the original and new formulations and was cumbersome and time-consuming. (Once the property was reformulated, however, the performance of INCA on this decomposed system was considerably better than that of the other tools. By $n = 30$ the INCA time was roughly an order of magnitude better than the times for the other tools and INCA could verify the property for much larger values of n. The differences in performance of the tools on this property, for the two versions of the Chiron system, are typical of what we observed on other properties. The implications of this are discussed in [4].)

Using the cycle elimination algorithm described here, we were able to verify the original property directly, without reformulating it, for $2 \leq n \leq 70$. In this case, the number of variables in the original ILP system (for $n \geq 3$) is

$$(262n + \lambda(n))/3,$$

where $\lambda(n)$ is 207, 395, or 301, according as n is congruent modulo 3 to 0, 1, or 2, respectively. (This reflects the way we chose to have artists register for events as we scaled the number of events.) The number of constraints in the original system is

$$(137n + \kappa(n))/3,$$

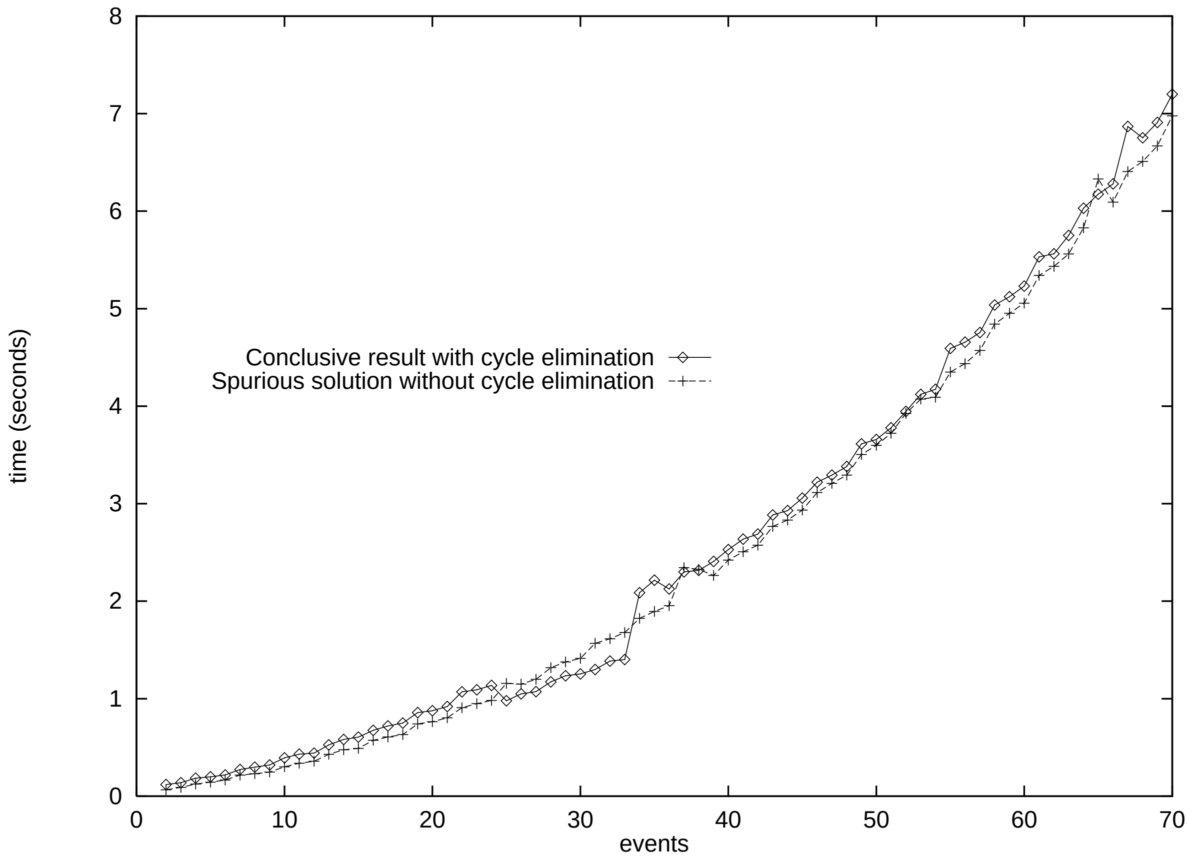

Fig. 7. CPLEX times for Chiron Property 4

where similarly the value of $\kappa(n)$ is 213, 301, or 257. For each n the graph $\hat{G}$ constructed from the `Dispatch_e1`-interval 2 sector of the flowgraph has 23 vertices and 63 edges; hence the algorithm adds 86 variables and 148 constraints. In this case, eliminating spurious cycles adds a constant number of variables and constraints as n increases. The CPLEX times for each n, for the original system for which CPLEX found a spurious solution and the result of the analysis was inconclusive, and for the augmented system for which the property was conclusively verified, are given in Figure 7. Again, the times are all under 8 seconds and represent a small portion of the total analysis time. (For $n = 70$, this was about 64 seconds.) As the figure shows, there is essentially no cost in additional CPLEX time for cycle elimination for this example.

C. The Cost of Unnecessarily Preventing Spurious Cycles

We also tried adding the cycle elimination variables and constraints to a system which already yielded a conclusive result. This might yield insight into the marginal cost of having INCA add

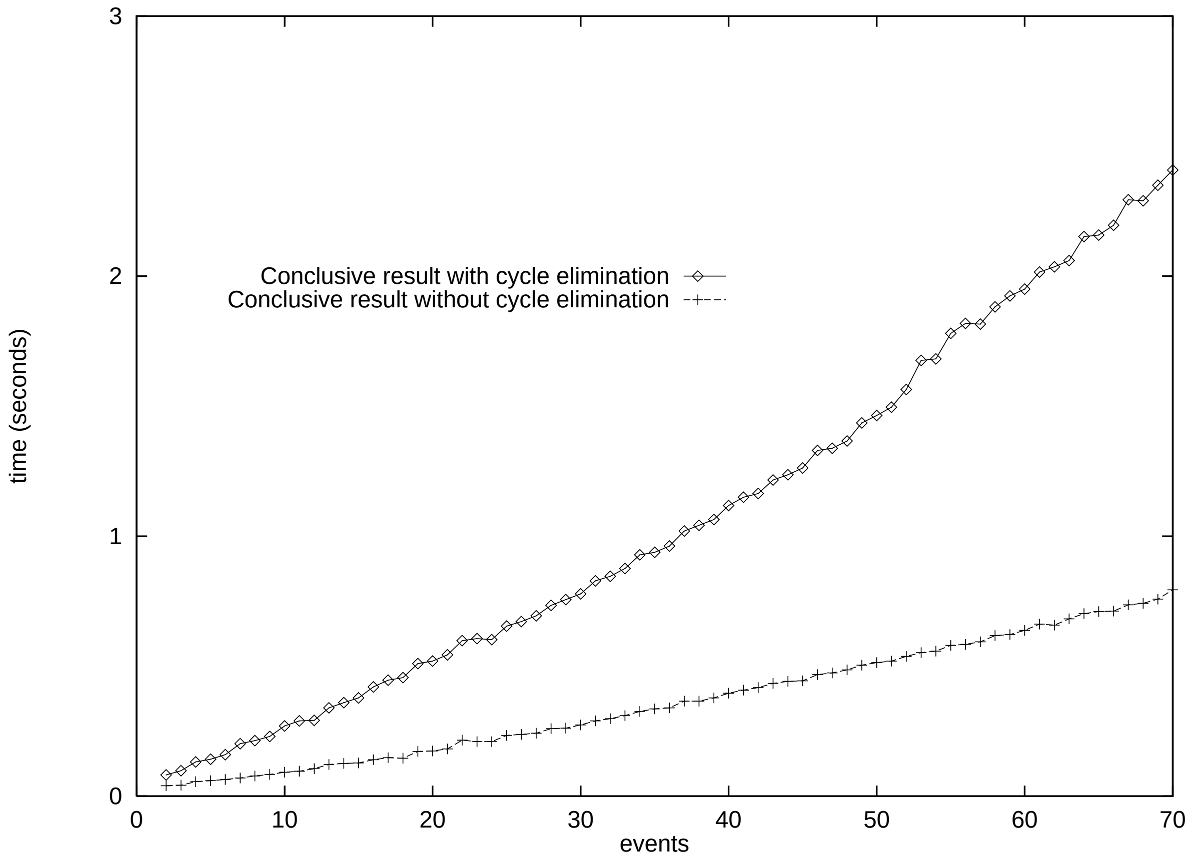

Fig. 8. CPLEX times for Chiron Property 1b

cycle elimination by default for any problem.

For this experiment, we used another property from [4]. In this case, we used Property 1b, which says that an artist never unregisters for an event unless it is already registered for that event. As in [4], we restricted ourselves to a single artist and event. The resulting property requires 2 intervals for its formulation as an INCA query. Using the decomposed dispatcher version of the client code, INCA verified this property without any need for cycle elimination, for $n \leq 70$. The number of variables in the INCA-generated ILP system (for $n \geq 3$) is

$$100n + \alpha(n),$$

where $\alpha(n)$ is 77, 146, or 107 according as n is congruent modulo 3 to 0, 1, or 2, respectively. The number of constraints is

$$51n + \beta(n),$$

where similarly $\beta(n)$ is 69, 96, or 81.

We then applied the cycle elimination algorithm to the entire flowgraph, which consists of 2 intervals of $n + 6$ tasks each. (In the experiment discussed in the previous section, we only applied the algorithm to a single task-interval sector of the flowgraph.) The flowgraph has

$$(124 + \gamma(n))/3$$

vertices, where $\gamma(n)$ is 204, 272, or 238, and

$$111n + \delta(n)$$

edges, where $\delta(n)$ is 116, 187, or 148. Hence cycle elimination adds

$$(457n + \mu(n))/3$$

new variables to the system, where $\mu(n)$ is 552, 833, or 682, and adds

$$(790n + \nu(n))/3$$

new constraints, where $\nu(n)$ is 897, 1391, or 1123. The times required by CPLEX to find the conclusive result in each case are graphed in Figure 8.

Although the ILP systems in the augmented case are quite large (18,087 variables and 22,563 constraints for $n = 70$) for the larger n, it still appears that CPLEX can determine the inconsistency of the system in a very short time (less than 3 seconds). For this example, the real cost in introducing cycle elimination in INCA lies in generating the new ILP system, not in solving it. (For $n = 70$, our Java program took about 28 seconds to generate the augmented ILP problem that eliminates cycles.)

D. An Example with Many Cycles

For each $n \geq 2$ we describe a concurrent system which we call Relay(n). This system has $n + 1$ tasks. The first task, `resource`, has a single variable which can take on any value from 0 to $n - 1$, and starts with the initial value $n - 1$. Within an infinite loop, it has entries for both setting the value of the variable, and getting the value. The remaining n tasks are called `ti`, for $i = 0, \ldots, n - 1$. Task `ti` does the following within an infinite loop: it first calls the entry in `resource` to get the value of the variable. It then checks to see if this value is equal to i, and if

```
package relay is
  subtype val is natural range 0..2;
  task resource is
    entry set (i : in val);
    entry get (j : out val);
  end resource;
  task t0;
  task t1;
  task t2;
end relay;
```

```
package body relay is
  task body resource is
    x : val := 2;
  begin
    loop
      select
        accept set (i : in val) do
          x := i;
        end set;
      or
        accept get (j : out val) do
          j := x;
        end get;
      end select;
    end loop;
  end a;

  task body t0 is
    y : val;
  begin
    loop
      resource.get (y);
      if y = 0 then
        resource.set (1);
      end if;
    end loop;
  end t0;
end relay;
```

```
task body t1 is
  y : val;
begin
  loop
    resource.get (y);
    if y = 1 then
      resource.set (2);
    end if;
  end loop;
end t1;

task body t2 is
  y : val;
begin
  loop
    resource.get (y);
    if y = 2 then
      resource.set (0);
    end if;
  end loop;
end t2;
```

Fig. 9. Source code for Relay example

so, it calls the entry in `resource` to set the value to $i + 1$ (if $i < n - 1$) or 0 (if $i = n - 1$). The source code for Relay(3) is given in Figure 9.

The intended behavior of this system is that the variable will be set to the following values in order: $0, 1, 2, ..., n - 1, 0, 1, 2, ..., n - 1, ...$, and so on. One property we checked is the following: if the variable is set to $n - 1$ then it must have previously been set to 0. Let P be the event in which task `resource` accepts a call to `set` with parameter 0, and let R be the event in which `resource` accepts a call to `set` with parameter $n - 1$. The standard query describing a violation

n	original variables	original constraints	new variables	new constraints	time w/o cycle elim	time with cycle elim
3	23	19	28	45	0.03	0.05
4	39	26	52	86	0.03	0.05
5	59	33	84	141	0.04	0.12
6	83	40	124	210	0.04	0.56
7	111	47	172	293	0.04	2.93
8	143	54	228	390	0.04	49.14
9	179	61	292	501	0.05	717.16

Fig. 10. Performance on the Relay Example

of this property consists of one interval which begins with the start of execution, ends with R and forbids P. (This property is an instantiation of the "Existence of P before R" pattern; see [2].)

The sector of the flowgraph for task `resource` has an enormous number of cycles; in fact, there are precisely $(2^n + 1)^{(n-1)} - 1$ distinct subsets of the vertex set which form cycles. (For $n = 6$, this is 1,160,290,624, and Relay(6) is the example mentioned in Section III-A.) It is not surprising, then, that of our various examples, this one posed the biggest challenge to cycle elimination.

For $n = 2$, INCA was able to verify the property in its preprocessing stage, without calling CPLEX. For $3 \leq n \leq 9$, we obtained a spurious solution with a cycle when we analyzed the INCA-produced system, and we were able to conclusively verify the property after applying cycle elimination to task `resource`. For the cycle elimination runs, we told CPLEX to give higher priority to the new s_e and d_v variables in its branch-and-bound strategy, and we tightened the integrality tolerance from 1E-05 to 1E-07. The data on the numbers of variables and constraints, and the time (in seconds) it took CPLEX to reach the spurious solutions and conclusive results are given in Figure 10. Note that for $n = 9$ it took a substantial amount of time—just under 12 minutes—for CPLEX to reach the conclusive result, although, as mentioned earlier, the time to produce the new variables and inequalities was less than 2 seconds.

V. R W

The most common method for detecting faults in computer systems is testing, that is, executing the system with a particular set of inputs and comparing that execution with the expected result. The main problem with testing, of course, is coverage: it is almost always infeasible to check more than a very small fraction of the possible executions of the system, and testing can give no information about executions that are not examined. Testing can thus miss serious faults. Testing is especially problematic for concurrent systems because such systems tend to behave nondeterministically in that the same inputs may lead to very different executions, depending on the order in which events occur in the different parts of the system. This can make it difficult even to reproduce a particular execution, and means that a test result in which the execution with particular input data matches the expected behavior does not even imply that the system will always behave correctly when it receives the same inputs.

Finite-state verification techniques, such as model checking [15], algorithmically check properties of a finite-state model of the system. By constructing models that represent all possible executions of the system, finite-state verification techniques can check whether a property such as freedom from deadlock, mutually exclusive use of a resource, or guaranteed response to a request, holds on all possible executions of a system. When a property does not hold, most finite-state verification tools can provide the user with a "counterexample" showing how the property can be violated. Such counterexamples can be extremely useful in isolating and understanding the fault.

These techniques vary in the nature of the model, the formalism used to express the properties of interest, and the method used to determine whether the model satisfies the properties. At a conceptual level, most finite-state verification tools model the system as a graph whose nodes represent abstract states of the system and whose edges represent transitions between system states corresponding to events in the execution of the system. Executions correspond to paths through this graph and properties can be specified in a temporal logic (e.g., LTL [16] or CTL [17]) or as an automaton accepting sequences of states or events. Algorithms for determining whether the model satsifies the property can be based on methods for walking the graph of states, computing the intersection of automata, data flow techniques [14] and other approaches.

As noted earlier, the main obstacle to the application of finite-state verification techniques to concurrent systems is the fact that the number of reachable states may be exponential in the number of concurrent processes in the system. A number of approaches concentrate on constructing a compact model, for instance by taking advantage of symmetries of the system [15] or using abstraction to collapse states that do not need to be distinguished to check a given property. The Bandera toolset [18, 19], for example, provides facilities for slicing to remove parts of a program that are not relevant to the property to be checked and for applying a library of safe abstractions to reduce the size of the model. Bandera is intended to be used to construct compact models for a variety of finite-state verification tools. Other techniques, such as the partial order methods used in S [8], avoid constructing or examining states that are not needed to check a particular property. Symbolic model checkers, such as SMV [9], check properties using operations on sets of states which can be represented compactly by special data structures. As mentioned above, however, most of the questions we want to ask about concurrent computer systems are at least *NP*-hard (e.g., [20–22]), and no approach will avoid the state explosion problem completely.

INCA models executions of a computer system that violate a particular property as solutions to a system of linear equations and inequalities and uses integer linear programming techniques to determine whether the system of inequalities has any solutions. This model may overrepresent the set of actual executions, so that some solutions do not correspond to real executions of the program. Thus, if the system of equations and inequalities has no solutions, no execution exists that violates the property. If the system of equations and inequalities does have solutions, however, there need not be any executions of the computer system that violate the property; the solutions may be spurious. This approach does not need to explicitly represent each state of the computer system and can take advantage of techniques from linear algebra to improve the verification process.

Several authors have applied ideas similar to those used in INCA to systems modeled by Petri nets [23]. For example, Murata, Shenker, and Shatz [24] used integer programming methods to compute counts of transition firings that return a net to its original marking and then try to eliminate "spurious" counts that do not correspond to real firing sequences. More recently, Esparza and Melzer [25] show how to represent *traps* using inequalities in order to sharpen the basic approach.

Techniques for detecting or removing cycles from graphs are of considerable importance in a number of areas of computer science, such as online deadlock detection. We note, however, that the connections between such techniques and the algorithm presented in this paper are somewhat limited. In particular, our algorithm does not determine whether cycles are present in the flowgraph, and does not modify the flowgraph to remove any cycles. If we regard the system of equations and inequalities as a model of certain sets of paths through the FSAs corresponding to the processes in the program, our algorithm can be viewed as a way of refining that model so that it does not represent any collections of paths containing disconnected cycles. It does this entirely at the level of the system of equations and inequalities, and thus is only directly applicable in settings where paths in graphs can be represented as solutions to systems of inequalities.

VI. C F W

Some finite-state verification tools always provide a conclusive result on any problem they can analyze. A tool that walks a graph of the reachable states of a concurrent system will never report that the system might deadlock when in fact the system is deadlock-free (assuming, of course, that the graph correctly represents the reachable state space of the system). But such a tool must be able to store the full set of reachable states, and is unable to report any results for a system whose reachable state space exceeds the storage available. Other tools, such as INCA, deliberately overestimate the collection of possible executions of the system, and thus accept the possibility of inconclusive results (or spurious reports of the possible faults), in order to increase the range of systems to which they can be applied.

For INCA, there are two main sources of imprecision in the representation of executions of the system. The first of these is the fact that semantic restrictions on the order of occurrence of events in different concurrent processes are generally not represented in the equations and inequalities used by INCA. The second source of imprecision is the fact that the equations and inequalities allow solutions in which the flow in the FSA representing a concurrent process may have cycles not connected to the initial state. In this paper, we have shown how imprecision caused by this second source may be eliminated. Although our method is aimed at improving the precision of INCA, it may also be relevant to other applications of integer programming involving flow networks.

Specific cases of inconclusive results can often be addressed by careful reformulation of the property being checked, although this may require the verification of additional properties to justify the reformulation. This process can require very substantial effort on the part of the human analysts, as well as considerable costs to carry out the necessary verifications. We have also sometimes addressed inconclusive results by manually inserting special inequalities to prevent disconnected flow on a small number of specific cycles. The problem with generalizing this approach is that the number of cycles may well be exponential in the size of the concurrent system, and each of the cycles requires a separate inequality. Even if it were feasible to automate the generation of these inequalities, the resulting ILP problems would be far too large to solve. The numbers of new variables and inequalities introduced by the method presented in this paper are linear in the number of states and transitions in the FSAs representing the processes of the concurrent system being analyzed.

We have reported here the results of some preliminary experiments aimed at assessing the cost, in increased time to solve the systems of equations and inequalities, of applying our method. These experiments suggest that the cost is relatively small, especially when the effort of the human analysts is taken into account. We plan to carry out additional experiments of the same type, and to integrate our technique into the INCA toolset so that we can also evaluate the time needed to generate the additional variables and inequalities more precisely.

We are also investigating approaches to eliminating some of the imprecision caused by not representing restrictions on the order of events in different processes. Fully representing the restrictions imposed by the semantics of the programming language or design notation may not be practical and might limit the applicability of INCA in the same way that having to store the full set of reachable states limits the applicability of tools based on exploring the graph of reachable states. We are therefore exploring methods that allow the analyst to control the degree to which restrictions on order are represented. For example, one approach that we are considering is to formulate some of the flow and communication equations in such a way that they hold at every stage of an execution, not just the end. These reformulated flow and communication equations therefore enforce some of the restrictions on the order of events in different processes. They also determine a region in n-dimensional Euclidean space, where n is the number of variables in the system of equations and inequalities. We then look for a point satisfying the full system of equations and inequalities that can be reached by taking certain integer-sized steps through this region. Successfully reducing this kind of imprecision will be important in applying the INCA approach to many systems where interprocess communication is only through access to shared data.

R

[1] J. C. Corbett and G. S. Avrunin, "Using integer programming to verify general safety and liveness properties," *Formal Methods in System Design*, vol. 6, pp. 97–123, January 1995.

[2] "Specification patterns web site." `http://www.cis.ksu.edu/santos/spec-patterns/`.

[3] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *Proceedings of the Twenty-First International Conference on Software Engineering*, (Los Angeles), pp. 411–420, may 1999.

[4] G. S. Avrunin, J. C. Corbett, M. B. Dwyer, C. S. Păsăreanu, and S. F. Siegel, "Comparing finite-state verification techniques for concurrent software," Tech. Rep. UM-CS-1999-069, Department of Computer Science, University of Massachusetts Amherst, Nov. 1999.

[5] A. T. Chamillard, L. A. Clarke, and G. S. Avrunin, "An empirical comparison of static concurrency analysis techniques," Tech. Rep. 96-84, Department of Computer Science, University of Massachusetts, 1996. Revised May 1997.

[6] J. C. Corbett, "An empirical evaluation of three methods for deadlock analysis of Ada tasking programs," in *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)* (T. Ostrand, ed.), (Seattle, WA), pp. 204–215, ACM Press (Proceedings appeared as a special issue of *Software Engineering Notes*), Aug. 1994.

[7] J. C. Corbett, "Evaluating deadlock detection methods for concurrent software," *IEEE Trans. Software Engineering*, vol. 22, pp. 161–179, Mar. 1996.

[8] G. J. Holzmann, "The model checker S ," *IEEE Trans. Software Engineering*, vol. 23, pp. 279–295, May 1997.

[9] K. L. McMillan, *Symbolic Model Checking*. Boston: Kluwer Academic Publishers, 1993.

[10] J. C. Corbett and G. S. Avrunin, "A practical method for bounding the time between events in concurrent real-time systems," in *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)* (T. Ostrand and E. Weyuker, eds.), (Cambridge, MA), pp. 110–116, ACM Press (Proceedings appeared in *Software Engineering Notes*, 18(3)), June 1993. An updated version is available for anonymous ftp on ext.math.umass.edu.

[11] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated derivation of time bounds in uniprocessor concurrent systems," *IEEE Trans. Software Engineering*, vol. 20, pp. 708–719, Sept. 1994.

[12] J. C. Corbett and G. S. Avrunin, "Towards scalable compositional analysis," in Wile [26], pp. 53–61.

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

[14] M. B. Dwyer and L. A. Clarke, "Data flow analysis for verifying properties of concurrent programs," in Wile [26], pp. 62–75.

[15] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model Checking*. MIT Press, 2000.

[16] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. New York: Springer-Verlag, 1992.

[17] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Programming Languages and Systems*, vol. 8, pp. 244–263, April 1986.

[18] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng, "Bandera : Extracting finite-state models from Java source code," in *Proceedings of the 22nd International Conference on Software Engineering*, pp. 439–448, June 2000.

[19] M. B. Dwyer, J. Hatcliff, R. Joehanes, S. Laubach, C. Păsăreanu, and H. Zheng, "Tool-supported program abstraction for finite-state verification," in *Proceedings of the 23rd International Conference on Software Engineering*, (Toronto, Canada), pp. 177–187, May 2001.

[20] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Informatica*, vol. 19, pp. 57–84, 1983.

[21] J. Reif and S. Smolka, "The complexity of reachability in distributed communicating processes," *Acta Informatica*, vol. 25, no. 3, pp. 333–354, 1988.

[22] R. H. B. Netzer and B. P. Miller, "On the complexity of event ordering for shared-memory parallel program executions," in *1990 International Conference on Parallel Processing*, (St. Charles, IL), pp. II–93–II–97, Aug. 1990.

[23] T. Murata, "Petri nets: Properties, analysis and applications," *Proceedings of the IEEE*, vol. 77, pp. 541–580, Apr. 1989.

[24] T. Murata, B. Shenker, and S. M. Shatz, "Detection of Ada static deadlocks using Petri net invariants," *IEEE Trans. Software Engineering*, vol. 15, no. 3, pp. 314–326, 1989.

[25] J. Esparza and S. Melzer, "Verification of safety properties using integer programming: Beyond the state equation," *Formal Methods in System Design*, vol. 16, pp. 159–189, 2000.

[26] D. Wile, ed., *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, (New Orleans), ACM Press (Proceedings appeared in *Software Engineering Notes*, 19(5)), Dec. 1994.