In *ISSTA 00, Proc. of the ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis* (Mary Jean Harrold, ed.), 191–200. ©2000 ACM

Improving the Precision of INCA by Preventing Spurious Cycles*

Stephen F. Siegel
Department of Computer Science
University of Massachusetts
Amherst, MA 01003-4610
siegel@cs.umass.edu

George S. Avrunin
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003-4515
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.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

General Terms

Design, Reliability, Verification

Keywords

INCA, finite-state verification, cycles, integer programming

1. INTRODUCTION

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 infeasible—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 3 explains our technique for improving INCA's precision, and the fourth section presents some preliminary data on its application. The final section summarizes the paper and discusses other issues related to the precision of INCA.

2. INCA

A complete discussion of the INCA approach, along with a careful analysis of its expressive power, is contained in [8]. In this paper, we will use a small (and quite contrived) example to sketch the basic INCA approach and show how certain

*Research partially supported by the National Science Foundation under grant CCR-9708184. The views, findings, and conclusions presented here are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the National Science Foundation, or the U.S. Government.

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 [8].

2.1 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 segment 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 segments of an execution. Suppose that an analyst wants to show that event b can never be preceded by 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 segment running from the start of the execution until the occurrence of a b, with the requirement that an a occur somewhere in the segment. (It could also be specified as a sequence of two segments, the first running from the start of the execution until an occurrence of an a, and the second starting immediately after the first and ending with a b. The former specification is generally more efficient, but the latter may provide additional precision in some cases. See Section 2.2.) INCA provides a query language allowing the analyst to specify various aspects of the segments (called "intervals" in the INCA query language) of execution.

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 segments 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` begins by rendezvousing 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`.

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

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 t2 is
end t2;

task t3 is
end t3;

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

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

Figure 1: A small example

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.

Suppose that we wish to check that an occurrence of a rendezvous at entry `b` cannot be preceded by a rendezvous at entry `a`. As described earlier, we may specify the violation as a segment 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 segment.

Since the segment ends with a rendezvous at the 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 segment. 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 segment implies that that FSA must be in state 6 at the end of the segment, 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 measur-

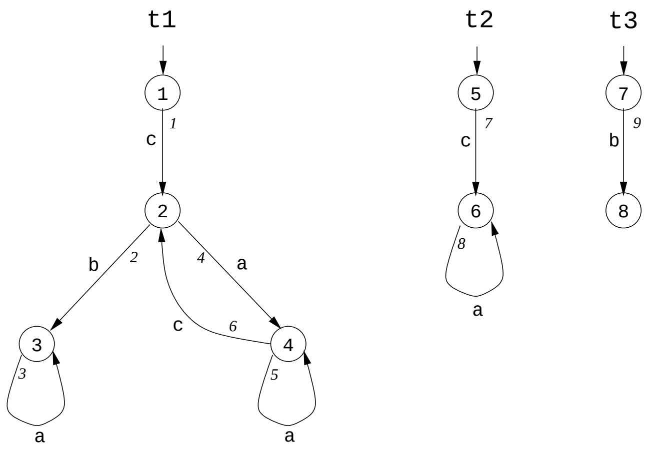

Figure 2: FSAs for example

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 $x_8 \geq 1$

Figure 3: System of equations and inequalities for example

ing 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

$$1 = x_1$$

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

$$x_9 = 1$$

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

$$x_3 + x_4 + x_5 = x_8$$

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

$$x_8 \geq 1$$

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 segment 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 INCA avoids enumerating the reachable states of the system and is inherently compositional, in the sense that that the equations and inequalities are generated from the automata corresponding to the individual processes, rather than from a single automaton representing the full concurrent 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 with other finite-state verification methods [2, 3, 4, 5] 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 SPIN and SMV. The INCA approach has also been extended to check timing properties of real-time systems [1, 6] and to prove trace equivalence of certain classes of systems [7].

2.2 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, they only represent necessary, not sufficient, conditions. A solution of the system of equations and inequalities may not correspond to an actual 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 segment of the execution, and do not directly impose restrictions on the order in which those events occur within the segment. 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 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 segments 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 in the final section of this paper.

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

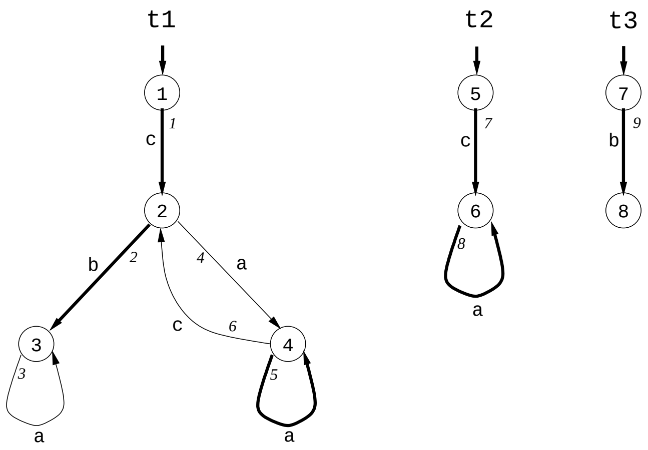

Figure 4: Solution with disconnected cycle

with some frequency in the analysis of more interesting systems. For instance, in our recent analysis of the Chiron user interface development system [2], we encountered solutions with disconnected cycles in trying to verify 2 of the 10 properties we checked. In those cases, we were able to reformulate the properties by specifying additional segments, verifying other properties that allowed us to eliminate some solutions, or choosing other events to represent the high-level requirement. These modifications, however, represent a considerable expense in increased analyst effort and verification time. In the next section, we describe a technique for eliminating these solutions with more than one component of flow in an FSA.

3. ELIMINATING SPURIOUS CYCLES

3.1 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 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

$$\sum_i x_{i,j} = 1 \tag{1}$$

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

$$\sum_{i \notin U, j \in U} x_{i,j} \geq 1, \tag{2}$$

which requires that the salesman travel from a city outside U to a city in U. (Of course, we need an inequality like (2) 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 (1) that 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

$$x_4 x_5 \geq x_5. \tag{3}$$

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 (3) with the linear inequality

$$x_5 - Bx_4 \leq 0. \tag{4}$$

The integer solutions of (3) having $x_4, x_5 \leq B$ are exactly the same as those of (4). (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 infeasible. A better method is required.

3.2 A More Practical Method

In this section, we describe a method for preventing spurious cycles that requires, for each FSA and segment of execution, $S+T$ new variables and $S+2T-1$ new inequalities, where S is the number of states in the FSA and T is the number of transitions.

The basic idea is essentially as follows. Suppose we have a solution to the system of equations and inequalities originally generated by INCA. For each FSA and each segment of execution, we attempt to construct a subgraph with the same vertices as the FSA but whose edges are a subset of those that have positive flow in the solution. We require that *(i)* if there is flow into a vertex v in the solution, some edge terminating in v 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 node is greater than that of any of its predecessors in the subgraph.

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.

We describe the procedure for generating this augmented system for the case of a single FSA F and a single segment of execution. For each variable x_i in the original system corresponding to a transition in F, we introduce a new variable s_i with bounds

$$0 \leq s_i \leq 1. \tag{5}$$

This variable will be 1 if the corresponding edge is in the subgraph, and 0 otherwise.

For each state v in F, we introduce a new variable d_v with bounds

$$0 \leq d_v \leq N, \tag{6}$$

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

We then generate inequalities involving these new variables. Each variable s_i corresponds to a transition from some state u of F to a state v. We generate the inequalities

$$x_i \geq s_i \tag{7}$$

$$d_v \geq d_u + (N+1)s_i - N. \tag{8}$$

The first inequality says that s_i must be 0 if x_i 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_i 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 $In(v)$ denote the number of transitions into the state v. For each state v of F, other than the initial state, we generate the inequality

$$B\,In(v) \sum_j s_j \geq \sum_j x_j, \tag{9}$$

where the sums are taken over all transitions into the state v and B is an upper bound on all the variables. (As noted earlier, B can be taken to be quite large.) If all the x_j are 0, this inequality is satisfied vacuously, but if any x_j is positive, the inequality forces some s_j to be positive. This means that, in a solution with flow into state v, some edge terminating in v belongs to the subgraph.

The argument sketched at the beginning of this section proves the following theorem, showing that this method eliminates only solutions with disconnected cycles.

THEOREM 1. *Let $\mathcal{P}$ be the system of equations and inequalities generated by INCA to check a particular property of a given concurrent system. Let $\mathcal{P}'$ be the augmented system constructed from $\mathcal{P}$ as described above. A solution of $\mathcal{P}'$ assigns values to all the variables in $\mathcal{P}$ as well as additional variables; we thus obtain an assignment of values to the variables in $\mathcal{P}$ from a solution to $\mathcal{P}'$ by projection. The set of integer solutions of $\mathcal{P}$ with all variables taking values at most B and no disconnected cycles is exactly equal to the set of projections of integer solutions of $\mathcal{P}'$ with all variables taking values at most B.*

In general, a query can specify more than one execution segment, so the situation is a bit more complicated. In the general case, INCA constructs a *flowgraph* as follows. First, it creates one copy of each FSA for each segment specified in the query. Each copy can then be optimized independently, removing unnecessary states or transitions, based on the restrictions imposed in the query for that segment. As seen in the example in Section 2.1, INCA can determine from the query the states in which each FSA could be at the end of each segment. It then adds a "connect" edge from each of the possible end states for segment i to the corresponding state in segment $i + 1$. These edges connect the flow representing events in one segment of an execution to flow in the next segment. Finally, an initial node is added with connect edges to certain states in the first segment of each task, and a final node is added with incoming connect edges from the possible end states in the final segment of each task. This flowgraph is the actual structure which INCA uses to generate the ILP system.

The algorithm described in this section can actually be applied to any subset of vertices in the flowgraph, rather than to the whole flowgraph, thereby eliminating only those spurious solutions in which there is a disconnected cycle contained in that subset. For given a subset W of vertices of the flowgraph, one can form a new graph V as follows. Create a vertex in V for each vertex in W, and also add an initial and a final vertex to V. For each edge joining two vertices in W, create a corresponding edge in V. For each edge originating outside W and terminating in W, create a corresponding edge in V from the initial vertex to the corresponding vertex. For each edge originating in W and terminating outside of W, create a corresponding edge in V from the corresponding vertex to the final vertex. Each edge in V has associated to it an ILP variable, which is the variable associated to the corresponding edge in the original flowgraph. So we can apply the algorithm to V, generating new variables and inequalities which are added to those INCA originally produced from the flowgraph, and the same arguments given above go through.

Restricting the algorithm in this way has many practical applications. Suppose, for example, that a solution contains a single disconnected cycle. It is clear that that cycle must lie within a single segment of a single task in the flowgraph. That is because there are no edges from a state in one segment to a state in a preceding segment, and there are no edges from states of one task to another. Now, to apply the cycle-elimination algorithm to the entire flowgraph might be very expensive, both in terms of the time and memory to generate the new variables and constraints, and the time and memory needed by the ILP tool to solve the new system. In this case, it makes sense to apply the algorithm only to the problematic segment of the problematic task. Typically, the segments behave quite independently, and the existence of spurious cycles in one segment is not related to the existence of spurious cycles in other segments.

One might be tempted to be as conservative as possible and apply the cycle-elimination algorithm to only those vertices involved in the offending cycle. This is usually fruitless, as, more often than not, another spurious solution will be found by expanding the cycle to include other vertices. However, no matter how much the cycle expands, it still must lie entirely in the single segment of the single task, and therefore the best strategy might be to apply the algorithm to the entire problematic segment in that task as soon as one spurious cycle appears there.

4. PRELIMINARY EXPERIMENTS

The current version of INCA consists of about 12,000 lines of Common Lisp. INCA writes out a file describing the system of equations and inequalities in a standard format (the MPS format), and we then 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 a 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. At this stage, however, we cannot measure how long it would take INCA to generate the augmented system of equations and inequalities.

For these experiments, we used INCA version 3.4, Harlequin Lispworks 4.1.0, and CPLEX version 6.5.1 on a Sun Enterprise 3500 with two processors and 2 GB of memory, running Solaris 2.6. 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 strategy nodeselect was set to 2, mip strategy branch was set to 1, and mip limits solutions was set to 1. (The first two affect choices made in the branch-and-bound algorithm 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 `time` command, and include both user and system time.

4.1 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 2.1. Given an integer $n \geq 1$, we modified the Ada program in Figure 1 to have n copies of task `t2` and to have $n+1$ alternatives in the outer `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 a rendezvous at entry `a` can never precede a rendezvous at entry `b`. INCA constructs an FSA for `t1` in which there are $2n+4$ nodes and $4n^2+3$ edges. (The picture is slightly different from what one might expect because we have added a start vertex and an end vertex, and INCA performs some trimming of the FSA.) There are $2^n + n - 1$ distinct subsets of the vertex set for `t1` which form cycles.

For each n, INCA finds a spurious solution involving a disconnected cycle in `t1`. Applying the algorithm in Section 3.2 to the portion of the flowgraph coming from the FSA for task `t1`, however, yields an ILP problem that CPLEX reports has no integer solutions, thus verifying that an `a` can never precede a `b`.

For $n \geq 3$, the number of variables in the INCA-generated ILP system is $4n^2+2n$, and the number of constraints (equations and inequalities) is $5n+1$. The number of variables in the new system is

$$(4n^2 + 2n) + (4n^2 + 2n + 7) = 8n^2 + 4n + 7,$$

and the number of constraints is

$$(5n + 1) + (8n^2 + 2n + 9) = 8n^2 + 7n + 10.$$

The time that it takes CPLEX to find a spurious solution to the original system and the time it takes to determine the inconsistency of the augmented system are shown in Figure 5. These times are very modest, all under 10 seconds, and are in fact dwarfed by the time it takes INCA to generate its internal representations of the problem and the original ILP system. (For $n = 30$, this was about 30 minutes.) It seems, however, that for large n, the substantial increase in the number of constraints in the augmented system, due to the large number of edges in the FSA for `t1`, does begin to have a significant impact on the time to solve the ILP problem.

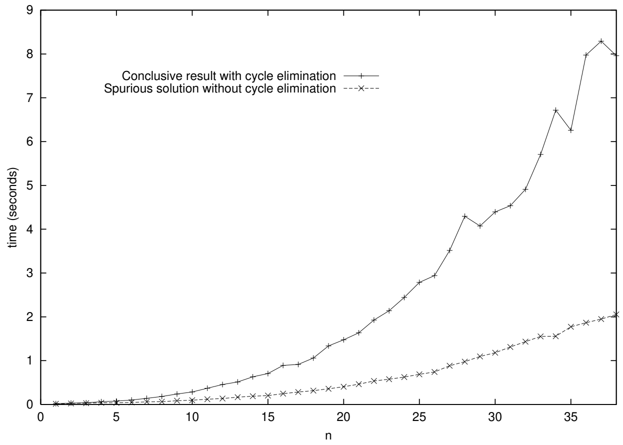

Figure 5: CPLEX times for scaled simple example

4.2 Spurious Cycles in Chiron

The second experiment involves the Chiron user interface system [9]. 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 *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 [2], we compared the performance of several finite-state verification tools (FLAVERS, INCA, SMV, and SPIN) 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 [2], 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 event `e2` until it has notified the appropriate artists of `e1`. To formulate this property as an INCA query takes 2 segments.

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 SPIN 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 `e`i. 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. While in general this decomposition greatly improves the performance of INCA, for this property INCA yields an inconclusive result. The problem is a disconnected cycle in the task *Dispatch_e*1 in the second segment.

To get around this problem, we reformulated the property using different events to represent the high-level property. 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 already 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 [2].)

Using the cycle elimination algorithm described here, we were able to verify the original property directly, for $2 \leq n \leq 70$. In this case there are 23 nodes and 63 edges in the problematic task/segment for all n. Hence for each n our algorithm adds 86 variables and 148 constraints to the ILP system. For $n \geq 3$, the number of variables in the original system is

$$82n + \lambda(n),$$

where $\lambda(n)$ is 58, 118, or 84, 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 up the number of events.) The number of constraints in the augmented system is

$$(133n + \kappa(n))/3,$$

where similarly the value of $\kappa(n)$ is 195, 281, or 235. 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 6. Again, the times are all under 5 seconds and represent a very small portion of the total analysis time. (For $n = 70$, this was about 2.5 minutes.) The spike at $n = 55$ in the CPLEX time for the augmented system seems to be due to the occurrence of certain numerical problems for this particular system.

4.3 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 cycle elimination by default for any problem.

For this experiment, we used another property from [2]. In

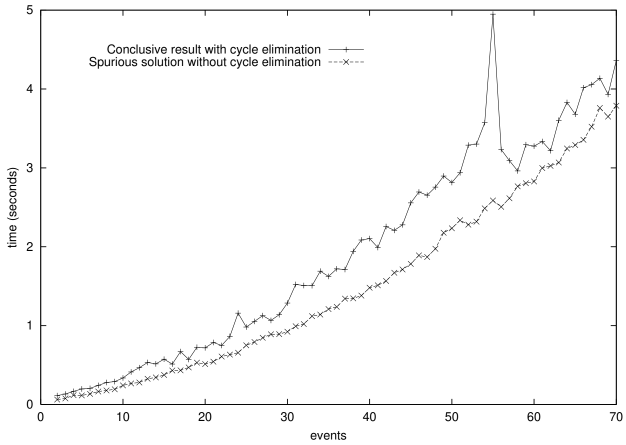

Figure 6: CPLEX times for Chiron Property 4

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 [2], we restricted ourselves to checking this for a single artist and event. The resulting property requires 2 segments 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 all of the $n + 6$ FSAs (recall that there is a separate *Dispatch_ei* for each of the n event types) and both segments. (In the experiment discussed in the previous section, we only applied the algorithm to one FSA and one segment.) This entailed adding

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

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

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

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 7.

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 4 seconds). If this example is typical, the real cost in introducing cycle elimination in INCA might lie in generating the new ILP system, not in solving it.

5. CONCLUSIONS AND FUTURE WORK

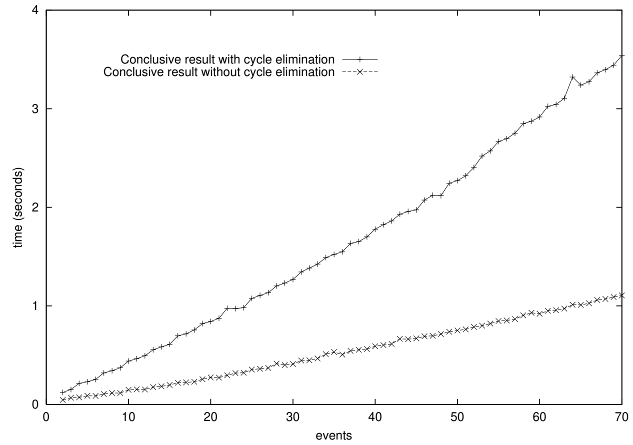

Figure 7: CPLEX times for Chiron Property 1b

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.

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 amounts of 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.

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.

6. REFERENCES

[1] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated derivation of time bounds in uniprocessor concurrent systems. *IEEE Trans. Softw. Eng.*, 20(9):708–719, Sept. 1994.

[2] 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 UM-CS-1999-069, Department of Computer Science, University of Massachusetts Amherst, Nov. 1999. URL: `http://ext.math.umass.edu/~avrunin/recent_pubs/comparing.ps`.

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

[4] J. C. Corbett. An empirical evaluation of three methods for deadlock analysis of Ada tasking

programs. In T. Ostrand, editor, *Proceedings of the 1994 International Symposium on Software Testing and Analysis (ISSTA)*, pages 204–215, Seattle, WA, Aug. 1994. ACM Press (Proceedings appeared as a special issue of *Software Engineering Notes*).

[5] J. C. Corbett. Evaluating deadlock detection methods for concurrent software. *IEEE Trans. Softw. Eng.*, 22(3):161–179, Mar. 1996.

[6] J. C. Corbett and G. S. Avrunin. A practical method for bounding the time between events in concurrent real-time systems. In T. Ostrand and E. Weyuker, editors, *Proceedings of the 1993 International Symposium on Software Testing and Analysis (ISSTA)*, pages 110–116, Cambridge, MA, June 1993. ACM Press (Proceedings appeared in *Software Engineering Notes*, 18(3)). An updated version is available at `http://ext.math.umass.edu/~avrunin/recent_pubs/issta_update.ps`.

[7] J. C. Corbett and G. S. Avrunin. Towards scalable compositional analysis. In D. Wile, editor, *Proceedings of the Second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 53–61, New Orleans, Dec. 1994. ACM Press (Proceedings appeared in *Software Engineering Notes*, 19(5)).

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

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