Formal Methods in System Design, 6, 97–123 (1995)
© 1995 Kluwer Academic Publishers, Boston. Manufactured in The Netherlands.

Using Integer Programming to Verify General Safety and Liveness Properties

JAMES C. CORBETT corbett@hawaii.edu
University of Hawaii at Manoa

GEORGE S. AVRUNIN avrunin@math.umass.edu
University of Massachusetts at Amherst

Abstract. Analysis of concurrent systems is plagued by the state explosion problem. We describe an analysis technique that uses necessary conditions, in the form of linear inequalities, to verify certain properties of concurrent systems, thus avoiding the enumeration of the potentially explosive number of reachable states of the system. This technique has been shown to be capable of verifying simple safety properties, like freedom from deadlock, that can be expressed in terms of the number of certain events occurring in a finite execution, and has been successfully used to analyze a variety of concurrent software systems. In this paper, we extend the technique to the verification of more complex safety properties that involve the order of events and to the verification of liveness properties, which involve infinite executions.

Keywords: Concurrent systems, automated verification, integer programming, safety, liveness

1. Introduction

Many concurrent systems can be modeled as a set of communicating finite state machines. In theory, this allows properties of such systems to be verified automatically by state enumeration and model checking techniques. In practice, however, the analysis of these systems is generally intractable since the number of system states grows exponentially with the number of state machines. This is commonly known as the *state explosion problem.*

Many techniques have been proposed to cope with this problem. Symbolic model checking techniques [6][15] use binary decision diagrams (BDDs) [5], a compact representation for boolean functions, to represent the state space of a system symbolically. BDDs can compactly represent certain kinds of regularity in the state space of a concurrent system and drastically improve the performance of model checking on these systems, though in general the size of the BDDs can grow as fast as the number of states in the system. These techniques have been applied to certain kinds of circuits, some standard concurrency problems such as the dining philosophers, and recently to a cache coherence protocol.

Partial order techniques [13][18][21] identify transitions that "commute" (have the same effect if performed in either order) and use this information to reduce the number of states explored by not differentiating equivalent interleavings of transitions. These techniques excel at analyzing systems in which the state explosion results primarily from interleaving the actions of largely independent processes. For example, these techniques explore only $O(n)$ states in a standard dining philosophers system of n philosophers. In a slight variation of this system in which a "host" process is added to prevent the deadlock, however, the number of states explored remains exponential in n.

Abstraction techniques [8] use homomorphisms that map the state space of the system to a smaller, more abstract transition system sufficient for verifying a particular property. The chief difficulty with these techniques is in finding a good abstraction, i.e., one that preserves just enough of the details of the system to verify a particular property. Although not completely automatic, these techniques have been applied to systems having over 10^{1300} states.

Compositional approaches [7][20][23], usually based on a *process algebra* [4][14][17], reduce the complexity of the analysis by composing the components of a concurrent system in stages and hiding internal details of the composed entity after each stage. These techniques are best applied to well-structured systems with simple interfaces between the subsystems. Automated tools for performing these analysis [10][22] have been used to verify a range of concurrent systems from simple network protocols such as the alternating bit protocol to standard concurrency problems such as the dining philosophers and the self-service gas station.

The inequality necessary condition method [2][3] avoids the enumeration of a system's states altogether. Given a concurrent system and a property to be verified, this method generates a system of linear inequalities that represents necessary conditions for the existence of an execution of the concurrent system violating the property. The inequalities express constraints on the number of times certain events can occur in relation to other events. The consistency of these necessary conditions is then checked using integer linear programming (ILP) methods. If the inequality system has no integral solutions, then the necessary conditions for the violation of the property cannot be satisfied, proving that the concurrent system has the property. This method has been automated and applied to some concurrent systems having as many as 10^{47} reachable states [3]. Unfortunately, the types of properties that can be verified by this method are somewhat limited. For example, it can verify that a system is free from deadlock, but it cannot verify liveness properties, which involve reasoning about infinite executions, nor can it directly verify properties like mutual exclusion, which involve the relative order of the events in an execution, rather than just the number of occurrences of these events.

In this paper, we extend the inequality necessary condition method to handle both infinite executions and properties involving the relative order of events. A further extension of these ideas in [11] enables the technique to verify properties expressible in linear time temporal logic, thus allowing a very general class of questions about a system to be answered while avoiding the construction of an exponentially-sized state graph. In the next section, we describe the model on which the method is based and outline the basic inequality necessary condition technique. The third section discusses the expressiveness of our extended analysis technique, and the fourth and fifth describe it in detail. We then report on some preliminary experiments demonstrating the feasibility of the technique and present some conclusions.

2. Model and Basic Technique

We model a concurrent system as a collection of coupled finite state automata (FSAs) with additional restrictions expressed as a set of recursive languages on the alphabets of the FSAs. The acceptance of a symbol by an automaton represents the occurrence of an event in the concurrent system. An event may represent a normal action of a component, such as initiating a communication with another component, or an error, such as waiting forever for a communication that never takes place. An execution of the concurrent program is thus modeled by a string of event symbols.

Formally, we regard a concurrent system as a triple (M, R, T) where M is a set of FSAs $M_1, \ldots, M_n$ with alphabets $\Sigma_1, \ldots, \Sigma_n$, $\Sigma = \bigcup_i \Sigma_i$, R is a set of recursive *restriction* languages $R_1, \ldots, R_m$ with alphabets $A_1, \ldots, A_m$, $A_i \subseteq \Sigma$ for all i, and $T \subseteq \Sigma$ is a terminal alphabet. Let $\rho_A(s)$ denote the projection of string s onto alphabet A (i.e., symbols of s not in A are removed). Then a string $t \in T^*$ represents a legal behavior or *trace* of the concurrent system if there exists a string $s \in \Sigma^*$ with $\rho_T(s) = t$ where $\rho_{\Sigma_i}(s) \in L(M_i)$ for all i and $\rho_{A_j}(s) \in R_j$ for all j.

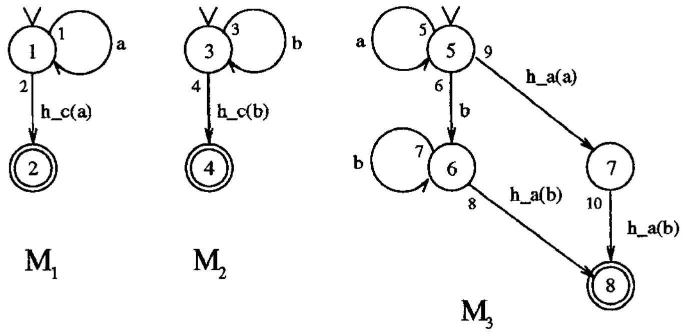
This model is general enough to represent many common communication mechanisms, including asynchronous message passing [2], but in this paper we will focus on the case where pairs of processes communicate synchronously over named channels that connect them. On each channel, one process acts as the *caller* while the other acts as the *acceptor*. We model such a communication using the channel name as an event symbol that appears in the alphabets of the FSAs of both processes. (We can also model the transmission of data across a channel by encoding the data into this symbol, but for simplicity, here we restrict communication to synchronization). The event that the process becomes permanently blocked waiting for communication over a channel is represented by the acceptance of a *hang symbol* for that channel. The hang symbols for channel e in the caller and acceptor processes are denoted *h_c(e)* and *h_a(e)* respectively. Since the two processes communicating over a particular channel cannot both become permanently blocked waiting for the other one to be ready to communicate, strings in which both *h_c(e)* and *h_a(e)* occur cannot correspond to executions of the concurrent system. The restriction language $\{h\_c(e),\ h\_a(e),\ \lambda\}$ excludes these strings. Figure 1 gives the specification for a small concurrent system in an Ada-like design language and shows the FSAs we would use to model that system. Note that the value of the variable `turn` has been encoded into the state of M_3. Also, in states that can engage in communication over multiple channels (e.g., state 5), there must be the option to accept a sequence of hang symbols: one for each channel. In all our examples, we shall take the set of event symbols appearing in an FSA or restriction language as its alphabet and not specify the alphabets separately.

The basic technique, detailed in [2], uses necessary conditions, in the form of linear inequalities, to either help find a trace with certain properties or prove that no such trace could exist. Every trace determines a path in each FSA from the starting state to an accepting state, representing the activities engaged in by the process corresponding to that FSA. The fact that the paths come from a trace implies that they satisfy certain conditions involving the interaction between processes and the constraints imposed by

```
task body one is
begin
  loop
    three.A;
  end loop;
end one;
```

```
task body two is
begin
  loop
    three.B;
  end loop;
end two;
```

```
task body three is
  turn : (one,two) := one;
begin
  loop
    select
      when (turn = one) =>
        accept A;
    or
      accept B;
      turn := two;
    end select;
  end loop;
end three;
```



$R_1 = \{h\_c(a), h\_a(a), \lambda\}$

$R_2 = \{h\_c(b), h\_a(b), \lambda\}$

Figure 1. Small Example

the restriction languages. Our technique finds flows in each FSA that meet weaker conditions. Specifically, it requires that, for each communication channel, the FSAs connected by that channel agree on the number of times that they communicated over that channel and that inequalities representing some aspects of the various restriction languages be satisfied.

To produce the inequalities, we assign a variable, x_i, called a *transition variable,* to each transition i that represents the number of times transition i is taken. We also assign an *accept variable,* f_i, to each accepting state i that will be one if the FSA containing state i is in that state at the end of the trace; otherwise it will be zero. First, we produce a *flow equation* for each state, equating the flow into the state with the flow out of the state (i.e., the number of times the state is entered equals the number of times it is exited). There is an implicit flow of one into the start state and accept variables are counted as flow out. We then produce a *communication equation* for each channel, equating the number of times the processes connected by the channel communicated over that channel. We also produce *restriction inequalities* to enforce the restriction languages which, in this case, simply forbid more than one hang symbol for each channel from occurring.

The system of inequalities generated thus far represents necessary conditions that will be satisfied by every trace of an execution of the concurrent system. To verify a particular property of the system, we add inequalities that represent the negation of that property. For example, to verify that the caller of channel e cannot become permanently blocked waiting for a communication on channel e, we would add the inequality $|h\_c(e)| \geq 1$, where $|h\_c(e)|$ is the number of occurrences of event *h_c(e)* in the trace (which is given by the sum of certain transition variables). Properties that can be verified are limited to those whose negations can be expressed by linear inequalities involving the numbers of occurrences of particular events.

The algorithm to generate the inequalities described above is shown in Figure 2. The abbreviations used in this and the other algorithms in this paper are summarized in Table 1. The notation $[expr]_{cond}$ indicates that the expression *expr* should be added to the inequality only if the condition *cond* is true, and otherwise zero should be added. The existence of every variable appearing in *expr* is always implicitly conjoined to this condition, which defaults to true if not specified. The inequalities generated by the algorithm for the example of Figure 1 are shown in Figure 3. Here, we have not specified a property to be verified; the inequality system shown represents necessary conditions for the existence of any (finite) trace.

These inequalities represent necessary conditions for an assignment of values to the transition variables to correspond to a trace. Clearly every set of paths corresponding to a trace will yield flows through the FSAs satisfying the communication and restriction inequalities, so these inequalities represent necessary conditions satisfied by every trace. The conditions are not sufficient, however, since not every set of flows satisfying the inequalities need correspond to a trace. There are several reasons for this. First, the communication equations do not guarantee that there is a consistent ordering of the communication events (e.g., one FSA could synchronously communicate with another over channel c and then channel d, while the other communicated over channel d and then channel c). Secondly, the presence of cycles in the FSAs can allow cyclic flows that

Table 1. Abbreviations used in Algorithms

accept(c)	transitions representing accept on channel c
alphabet(M)	alphabet of FSA M
call(c)	transitions representing call on channel c
fsa(i)	FSA containing state/transition i
hang_a(c)	transitions in acceptor representing blockage on channel c
hang_c(c)	transitions in caller representing blockage on channel c
in(j)	transitions into state j
label(k)	event symbol labeling transition k
out(j)	transitions out of state j
occur(e)	transitions labeled with event e in any one FSA containing e
SCC(M)	transitions in any strongly connected component of FSA M
start(j)	true if state j is a start state of an FSA, else false
states(M)	states of FSA M
trans(M)	transitions of FSA M

```
Input:   A set $M$ of FSAs
         A property $P$ to be verified
Output:  A set of inequalities

For each transition $k$ of an FSA of $M$:
    Create transition variable $x_k$
For each accepting state $j$ of an FSA of $M$:
    Create accept variable $f_j$
For each state $j$ of an FSA of $M$:
    Generate flow equation: $[1]_{start(j)} + \sum_{k \in in(j)} x_k = \sum_{k \in out(j)} x_k + [f_j]$
For each channel $c$:
    Generate synchronization equation: $\sum_{k \in call(c)} x_k = \sum_{k \in accept(c)} x_k$
    Generate restriction inequality: $\sum_{k \in hang\_c(c)} x_k + \sum_{k \in hang\_a(c)} x_k \leq 1$
Generate additional inequalities specifying the violation of $P$
```

Figure 2. Basic Algorithm

$$\begin{array}{rcll}
\textbf{Flow:} & & & \textbf{(state)} \\
1 + x_1 & = & x_1 + x_2 & (1) \\
x_2 & = & f_2 & (2) \\
1 + x_3 & = & x_3 + x_4 & (3) \\
x_4 & = & f_4 & (4) \\
1 + x_5 & = & x_5 + x_6 + x_9 & (5) \\
x_6 + x_7 & = & x_7 + x_8 & (6) \\
x_9 & = & x_{10} & (7) \\
x_8 + x_{10} & = & f_8 & (8) \\
\textbf{Communication:} & & & \textbf{(channel)} \\
x_1 & = & x_5 & (a) \\
x_3 & = & x_6 + x_7 & (b) \\
\textbf{Restriction:} & & & \textbf{(number)} \\
x_2 + x_9 & \leq & 1 & (1) \\
x_4 + x_8 + x_{10} & \leq & 1 & (2)
\end{array}$$

Figure 3. Inequality System for Basic Technique

are not connected to the path found within the FSA. For example, there can be a cyclic flow on arc 7 in M_3 of Figure 1 even if the flow from the start state passes through arcs 9 and 10; the flow equation for state 6 does not constrain the transition variable for arc 7. Finally, the restriction languages, in general, may not be simple enough to capture with linear inequalities. In such a case, we could enforce only certain aspects of the restriction language. For example, we might add the equation $|a| = |b|$ for the restriction language generated by $(ab)^*$, but this equation does not enforce the alternation of a and b. For these reasons, a solution to the inequality system may not correspond to a trace of the concurrent system. If such a solution arises, the analysis is inconclusive since the presence of that solution implies nothing about the existence of another solution that does correspond to a trace. In our experience [3], however, such spurious solutions are uncommon. Also, we can sometimes add additional inequalities to remove such solutions.

As shown in [2], the formalism and analysis technique can also be applied to systems that use an asynchronous communication mechanism. For such systems, the communication and restriction inequalities would be different. For example, the communication inequalities would require that the number of messages received along each channel not exceed the number of messages sent along that channel. A description of how the analysis technique and the extensions we present here can be adjusted for use with such systems is given in [11].

3. Expressiveness

In this paper, we extend the basic technique presented in the last section to the verification of properties whose negations can be specified by *ω-star-less*[1] expressions, which are ω-regular expressions [19] of the form:

$$\bigcup_{i=1}^{m} S_{i,1}^{*} e_{i,1} S_{i,2}^{*} e_{i,2} \ldots S_{i,n_i}^{*} e_{i,n_i} S_{i,n_i+1}^{*} T_i^{\omega}$$

where $S_{i,j} \subseteq \Sigma$, $e_{i,j} \in \Sigma$, $T_i \subseteq \Sigma$. Specifically, given an ω-star-less expression, the extended technique produces necessary conditions for the existence of a trace lying in the language of infinite strings generated by the expression. This extended technique relies on two key ideas. The first idea allows the technique to test for properties in which events occur in a specific order and is described in Section 4. The second idea allows the technique to deal with infinite traces and is described in Section 5. In this section, we discuss the power of the technique and some related techniques from [11] in relation to model checking. We then sketch how an analysis with our technique proceeds.

Model checking techniques can determine whether a system satisfies a formula of temporal logic. It is well known that such first order logics are equivalent in expressive power to *star-free* regular expressions [19], which allow concatenation, union, and negation, but not Kleene star. If we define *star-less* expressions as regular expressions of the form:

$$\bigcup_{i=1}^{m} S_{i,1}^{*} e_{i,1} S_{i,2}^{*} e_{i,2} \ldots S_{i,n_i}^{*} e_{i,n_i} S_{i,n_i+1}^{*}$$

then we see that star-less expressions are star-free (A^* for any $A \subseteq \Sigma$ is star-free—e.g., $\Sigma^* = \overline{\emptyset}$). Unfortunately, there exist languages that can be defined with star-free expressions that cannot be defined with star-less expressions. This follows from the strictness of the dot-depth hierarchy of star-free expressions and the observation that star-less expressions have bounded dot-depth. For the infinite case, first order logics are equivalent in expressibility to ω-regular expressions of the form:

$$\bigcup_{i=1}^{m} A_i B_i^{\omega}$$

where A_i, B_i are star-free expressions. Again, there exist languages definable with such expressions that are not definable using ω-star-less expressions (e.g., $(ab)^{\omega}$). Thus ω-star-less expressions are strictly less expressible than temporal logic. In practice, we have not yet encountered any commonly verified concurrency properties that are not expressible with ω-star-less expressions, though our experience with the technique is limited.

Two extensions of our technique are presented in [11]. The first allows arbitrary inequalities over the number of occurrences of certain events to be added to the inequality system generated for the analysis of an ω-star-less expression. These inequalities restrict attention to traces lying in the language of the ω-star-less expression and satisfying the constraints expressed by the inequalities (e.g., the number of a events equals the number of b events). With such inequalities, it is easy to define languages that cannot be defined with star-free expressions (e.g., strings with an even number of a events). On the other hand, the depth-1 Dyck language cannot be expressed with star-less expressions and inequalities, so the class of star-free languages and the class of languages that can be expressed with star-less expressions and inequalities are not comparable. The implementation of our technique for ω-star-less expressions allows these additional inequalities to be added and we have used them in several experiments, usually to reduce the size of the ω-star-less expression required.

The second extension described in [11] allows the verification of properties specified by a Büchi automaton. Since Büchi automata are more expressive than first order logic [19], this implies that the technique can be used for any property expressible in linear temporal logic. This extension relies on the same two ideas as the technique for ω-star-less expressions. We do not describe it here, however, because, unlike the technique for ω-star-less expressions, it has not yet been implemented and tried on sample systems; hence the quality of the necessary conditions it produces, and thus its practical significance, is not known.

Given a concurrent system and a property whose negation is expressible as an ω-star-less expression, we can apply the extended techniques presented here as follows. We produce necessary conditions, in the form of linear inequalities, for the existence of a trace of the concurrent system that is also generated by the ω-star-less expression. If these conditions are unsatisfiable (i.e., the inequality system has no integral solution), then there are no traces of the system violating the property, so the property must hold. If the conditions are satisfiable (i.e., the inequality system does have an integral solution), then the property may or may not hold. If the necessary conditions are strong, however, the property will usually not hold when the conditions are satisfiable. Our experience is that our necessary conditions are strong. Furthermore, if the property does not hold, a solution satisfying our necessary conditions can often be used to find a trace violating the property.

4. Queries Involving Order

In this section, we describe a technique for verifying more complex safety properties in which the order of the events involved is significant. Since a violation of a safety property can be shown by a prefix of a trace, we may use regular expressions to generate these prefixes and postpone dealing with infinite traces until we address liveness properties in Section 5. Here, we present a technique for verifying safety properties whose violations are expressible with star-less expressions.

The basic technique presented in Section 2 can easily find traces in which certain event symbols occur a specified number of times, but it cannot find traces in which these symbols occur in a specific order. For example, to find a trace with one a event and one b event in the system of Figure 1, we would add $x_1 = 1$ and $x_3 = 1$ to the inequality system in Figure 3. There does not appear to be any way, however, to add equations that require the events to occur in a specific order. This is a serious limitation since many safety properties (e.g., mutual exclusion) constrain only the order of events and not their number. To produce necessary conditions for a trace containing a specific sequence of events, we use those events to divide the trace into segments we call *intervals*, we produce an inequality system for each interval, and then we connect these inequality systems together. For brevity, we will use the term "interval" to refer to the segment of a trace, to the inequality system representing necessary conditions for the existence of this segment, as well as to solutions to this inequality system interpreted as flows through the FSAs (e.g., we might speak of "flows through interval i" rather than "flows through the FSAs represented by the solution to the inequality system generated for interval i").

We explain the technique using the example of Figure 1 and then present the algorithm. Suppose we want to verify that there are no a events after any b event. The negation of this property can be expressed by the star-less expression $\Sigma^* b(\Sigma - \{a, b\})^* a$. We produce necessary conditions for the existence of a prefix of a trace containing a b followed by an a, as generated by this expression. We divide the prefix into two intervals. The first interval is from the initial state of the system to the state of the system after the b event (generated by $\Sigma^* b$). The second interval is from the state of the system after the b event to the state of the system after the a event (generated by $(\Sigma - \{a, b\})^* a$). For each interval, we produce an inequality system similar to the one generated by the basic algorithm, but with the following differences. We want the inequality system for the first interval to find flows ending after a b event rather than at accepting states. To achieve this, we assign to each state j having an incoming b transition a *connection variable* $c_{1,j}$ that will be one if the FSA containing state j is in state j at the end of the first interval, and will be zero otherwise. In FSAs not containing b events, we assign connection variables to all states. Note that requiring the interval to end in an FSA at a state with an incoming b transition does not guarantee that a b event occurred in that FSA during the interval. Therefore, we add a *requirement inequality* stating that at least one b event occurs. Since we are seeking only a prefix of a trace, we do not assign accept variables. If the connection variables are counted as flow out in the flow equations, as accept variables are treated in the basic technique, then the resulting inequality system will find a flow in each FSA from a starting state to a state in which the FSA could be immediately after a b event. Furthermore, in FSAs with b events, the flow must pass through at least one such event.

The inequality system for the second interval must find a flow in each FSA from the state the FSA was in at the end of the first interval to a state in which the FSA could be after an a event. To each state j in which an FSA could be following an a event, we assign a connection variable, $c_{2,j}$, that will be one if the FSA is in that state at the end of the second interval, and will be zero otherwise. In this interval, there can be no b events and only one a event (at the end), so we produce requirement equations setting the number of occurrences of a to one and the number of occurrences of b to zero. We then count the connection variables from the first interval as flow in, rather than having an implicit flow in of one at the start states, and count the connection variables from the second interval as flow out. Finally, the restriction inequalities are produced as before and involve the number of hang symbols from both intervals.

$$
\begin{array}{rcll}
\textbf{Flow (interval 1):} & & & \textbf{(state)} \\
1 + x_{1,1} & = & x_{1,1} + x_{1,2} + c_{1,1} & (1) \\
x_{1,2} & = & c_{1,2} & (2) \\
1 + x_{1,3} & = & x_{1,3} + x_{1,4} + c_{1,3} & (3) \\
x_{1,4} & = & 0 & (4) \\
1 + x_{1,5} & = & x_{1,5} + x_{1,6} + x_{1,9} & (5) \\
x_{1,6} + x_{1,7} & = & x_{1,7} + x_{1,8} + c_{1,6} & (6) \\
x_{1,9} & = & x_{1,10} & (7) \\
x_{1,8} + x_{1,10} & = & 0 & (8) \\
\textbf{Communication (interval 1):} & & & \textbf{(channel)} \\
x_{1,1} & = & x_{1,5} & (a) \\
x_{1,3} & = & x_{1,6} + x_{1,7} & (b) \\
\textbf{Requirement (interval 1):} & & & \textbf{(symbol)} \\
x_{1,3} & \geq & 1 & (b) \\
& & & \\
\textbf{Flow (interval 2):} & & & \textbf{(state)} \\
c_{1,1} + x_{2,1} & = & x_{2,1} + x_{2,2} + c_{2,1} & (1) \\
c_{1,2} + x_{2,2} & = & 0 & (2) \\
c_{1,3} + x_{2,3} & = & x_{2,3} + x_{2,4} + c_{2,3} & (3) \\
x_{2,4} & = & c_{2,4} & (4) \\
x_{2,5} & = & x_{2,5} + x_{2,6} + x_{2,9} + c_{2,5} & (5) \\
c_{1,6} + x_{2,6} + x_{2,7} & = & x_{2,7} + x_{2,8} & (6) \\
x_{2,9} & = & x_{2,10} & (7) \\
x_{2,8} + x_{2,10} & = & 0 & (8) \\
\textbf{Communication (interval 2):} & & & \textbf{(channel)} \\
x_{2,1} & = & x_{2,5} & (a) \\
x_{2,3} & = & x_{2,6} + x_{2,7} & (b) \\
\textbf{Requirement (interval 2):} & & & \textbf{(symbol)} \\
x_{2,1} & = & 1 & (a) \\
x_{2,3} & = & 0 & (b) \\
& & & \\
\textbf{Restriction:} & & & \textbf{(number)} \\
x_{1,2} + x_{1,9} + x_{2,2} + x_{2,9} & \leq & 1 & (1) \\
x_{1,4} + x_{1,8} + x_{1,10} + x_{2,4} & & & \\
+ x_{2,8} + x_{2,10} & \leq & 1 & (2)
\end{array}
$$

Figure 4. Inequality System for Prefix of Trace Generated by $\Sigma^* b(\Sigma - \{a, b\})^* a$

The inequality system produced for the system in Figure 1 given the expression $\Sigma^* b(\Sigma - \{a, b\})^* a$ is shown in Figure 4. The transition variable for transition j of interval i is denoted $x_{i,j}$. The whole system finds a flow in each FSA starting at the start state, proceeding through the first interval to a state with a connection variable for b, and then continuing through the second interval to a state with a connection variable for a.

Note that this inequality system, which represents necessary conditions for the existence of a prefix of a trace generated by $\Sigma^* b(\Sigma - \{a, b\})^* a$, has no integral solution. This proves that no trace containing a b followed by an a exists. For this trivial example, an appropriate kind of intersection between M_3 and the automaton for ba could have shown this; however, the above technique can be applied even if the events a and b are in different FSAs, as shown by the first example in Section 6.

We now present the algorithm for generating inequalities representing necessary conditions for the existence of a prefix of a trace generated by a star-less expression. We first present the algorithm for generating such conditions for one disjunct of a star-less expression, which we call a *sequence*, and then describe how the conditions for several sequences can be combined to represent necessary conditions for the existence of a prefix of a trace generated by any one of the sequences. A sequence of a star-less expression is a regular expression of the form:

$$S_1^* e_1 S_2^* e_2 \dots S_n^* e_n S_{n+1}^*$$

The algorithm to generate necessary conditions for the existence of a prefix of a trace generated by a sequence is shown in Figure 5. If $S_{n+1} = \Sigma$, we may omit the last interval (iteration $i = n + 1$), as we did in the example above. The correctness of these conditions is proved in [11].

Given a set of sequences, we derive necessary conditions for the existence of a trace generated by their disjunction as follows. We assign a *sequence variable*, s_i, to each sequence and generate an equation summing the sequence variables to one. We generate an inequality system, which we call a *sequence system*, for each sequence. A sequence system is similar to the inequality system generated for a sequence, as described above, with the following exceptions:

- In the sequence system for sequence i, the implicit flow into the start states in the first interval is s_i rather than one.
- In the sequence system for sequence i, the requirement equations that require a symbol occur exactly (or at least) once are changed to require the occurrence of that symbol exactly (or at least) s_i times.

If the sequence variable for a sequence system is set to one, it is the same as the inequality system that would be generated for the sequence standing alone. If the sequence variable for a sequence system is set to zero, it will always have the trivial solution where all the variables are zero. By this construction, it is clear that there exists an integral solution to the inequality system for the disjunction if and only if there exists an integral solution to at least one of the inequality systems generated for the sequences. Thus the resulting

```
Input:   A set $M$ of FSAs
         A sequence: $S_1^* e_1 S_2^* e_2 \dots S_n^* e_n S_{n+1}^*$
Output:  A set of inequalities

For each interval $i = 1, \dots, n+1$:
    For each transition $k$ of an FSA of $M$:
        Create transition variable $x_{i,k}$
    For each state $j$ of an FSA of $M$:
        If $i = n+1$ or $e_i \notin \mathit{alphabet}(\mathit{fsa}(j))$ or $\exists k \in \mathit{in}(j)(\mathit{label}(k) = e_i)$ then
            Create connection variable $c_{i,j}$
For each interval $i = 1, \dots, n+1$:
    For each state $j$ of an FSA of $M$:
        Generate flow equation:
            $[1]_{i=1 \wedge start(j)} + [c_{i-1,j}] + \sum_{k \in in(j)} x_{i,k} = \sum_{k \in out(j)} x_{i,k} + [c_{i,j}]$
    For each channel $c$:
        Generate synchronization equation: $\sum_{k \in call(c)} x_{i,k} = \sum_{k \in accept(c)} x_{i,k}$
    For each $e$ in $\Sigma$:
        If $e \notin S_i$ then
            Generate requirement equation: $\sum_{k \in occur(e)} x_{i,k} = [1]_{i \neq n+1 \wedge e = e_i}$
        Else
            Generate requirement equation: $\sum_{k \in occur(e)} x_{i,k} \geq [1]_{i \neq n+1 \wedge e = e_i}$
For each channel $c$:
    Generate restriction inequality:
        $\sum_{i=1}^{n+1} \left( \sum_{k \in hang\_c(c)} x_{i,k} + \sum_{k \in hang\_a(c)} x_{i,k} \right) \leq 1$
```

Figure 5. Algorithm for Sequence

inequality system represents necessary conditions for the existence of a prefix of a trace generated by the disjunction of the sequences.

The size of the inequality system for the disjunction is equal to the sum of the sizes of the inequality systems for the sequences, plus one additional variable per sequence, plus one additional equation (summing the sequence variables to one). An optimization described in [11] can significantly reduce the size of the inequality system for a disjunction by having different sequences share the same transition variables.

We have described how to generate necessary conditions for the existence of a prefix of a trace generated by a star-less expression. In the next section, we describe how to generate necessary conditions for the existence of an infinite suffix of a trace and how to combine these two kinds of conditions to form necessary conditions for the existence of a potentially infinite trace generated by an ω-star-less expression.

5. Infinite Traces

In this section, we address the verification of liveness properties. Note that the basic technique presented in Section 2 does not admit infinite traces, i.e., traces in which one or more FSAs continue engaging in actions forever. For example, the inequality system in Figure 3 has no integral solution since all of the traces of the concurrent system are infinite (there is no way for all of the FSAs to reach accepting states without violating the restrictions). This limitation is serious since the violation of a liveness property must be shown by a complete trace, which may be infinite. Therefore, to verify liveness properties, we must extend the formalism to represent infinite traces and extend the analysis technique to generate necessary conditions for the existence of infinite traces. We may then use the inconsistency of these conditions as proof that no infinite trace exists that violates a particular liveness property, just as the inconsistency of the conditions generated by the basic technique was used to prove that no finite trace exists that violates a particular safety property.

We first extend the model of Section 2 to allow infinite traces and then describe how to generate necessary conditions for the existence of these traces. The model of Section 2 represents an execution of a concurrent system with a finite string. To represent infinite executions, we use infinite strings of event symbols and make two changes to our representation of a system. First, the processes of the system are modeled by Büchi automata [19] rather than FSAs. A Büchi automaton is the infinite analog of an FSA, accepting languages of infinite strings rather than finite strings. The only difference is the condition for acceptance. In an FSA, a computation must end in an accepting state for the string to be accepted. A Büchi automaton accepts an infinite string if and only if the infinite computation on that string enters at least one of the accepting states of the automaton infinitely often. Büchi automata accept ω-regular languages, the infinite analog of regular languages. The Büchi automaton, M_ω, used to model a process looks exactly like the FSA, M, used before except that:

- All states in M_ω are accepting. This admits any infinite path through the automaton as a legal trace of the process.

- To each state of M_ω that was accepting in M, we add a self-loop (i.e., a transition from the state to itself) on a *stopped symbol*, denoted s_M, unique to the automaton. This allows legal finite behaviors of processes within the framework of infinite strings: for every finite string t accepted by M there is an infinite string $t(s_M)^\omega$ accepted by M_ω.

The second change made to the formalism involves the restrictions. Each restriction language is defined to be the union of a recursive language of finite strings over Σ and an ω-context-free language [19] of infinite strings over Σ. Restriction languages must include both finite and infinite strings since the projection of an infinite string onto an alphabet may be either finite or infinite. This completes the extension of the formalism. The analysis technique will still use the FSA representation of the processes, but will treat them as Büchi automata where appropriate.

We now describe how to generate necessary conditions for the existence of infinite traces. Consider the simplest case where we are seeking any infinite trace of a concurrent system (as opposed to a trace with a specific property). We can always divide such a trace into a *finite interval*, containing all events occurring only finitely many times in the trace, and a *perpetual interval*, containing only events occurring infinitely often in the trace. Solutions to the inequalities generated for the perpetual interval represent finite flows through the FSAs satisfying certain consistency requirements, however, these flows are interpreted differently than those found by the previous techniques. Any finite flow through an arc in the inequality system generated for the perpetual interval represents the infinite repetition of the event labeling that arc. For example, the infinite trace

ababbcbbcbbcbbc...

might be represented by a flow through arcs labeled *aba* in the finite interval, and a flow through arcs labeled *bc* in the perpetual interval. Note that this representation does not capture any information about the order in which the events in the perpetual interval are repeated, nor their relative frequency of occurrence, but only *that* they are repeated infinitely often. This suffices to verify properties specified with ω-star-less expressions. The transitions in a particular FSA that occur infinitely often in the trace must be part of a strongly connected component (SCC) in the FSA when viewed as a graph (i.e., to run forever, the FSA must traverse a cycle or set of interconnected cycles). An isolated state is not considered connected to itself unless there is an explicit self-loop, so these SCCs must contain at least one arc.

We construct an inequality system for the finite interval and another for the perpetual interval and connect these systems together to form an inequality system representing necessary conditions for the existence of an infinite trace. The inequality system for the finite interval is similar to the inequality system generated by the basic technique of Section 2 except for the following: To each state j of an FSA that is part of a SCC, we assign a *perpetual variable*, p_j, that counts as flow out in the flow equations (just as the accept variables do). This variable will be one if the FSA repeats a set of transitions, starting from state j, infinitely often, and it will be zero otherwise. Accept variables are also generated for the finite interval. For efficiency, the analysis technique uses these variables to allow the termination of processes, rather than using the infinite repetition of a stopped symbol, as in the formalism.

The inequality system for the perpetual interval represents necessary conditions for the existence of an infinite suffix of a trace. In this interval, any positive flow through an arc represents infinitely many occurrences of the event labeling that arc in the trace. We generate the perpetual interval system as follows. First, we conceptually remove parts of the FSAs not part of SCCs (in the example of Figure 1, we remove transitions 2, 4, 6, 8, 9, and 10, and ignore states 2, 4, 7, and 8). Second, we generate flow equations for the remaining states, but we do not add an implicit flow of one into the start states, nor do we include any other types of variables (e.g., accept, perpetual, connection). The connection between the finite and perpetual intervals is different than the connection between intervals in the technique of Section 4, as we will explain. Third, we generate synchronization equations for each channel equating the number of perpetual calls and accepts for that channel. Since hang symbols cannot occur perpetually, the perpetual interval system contains no restriction inequalities. Requirement inequalities are added only if the set of events that can be repeated forever (i.e., the T_i sets in an ω-star-less expression) is a strict subset of Σ, in which case these inequalities set to zero the number of occurrences of events not permitted to occur in the infinite suffix. In the case where we seek any infinite trace, no requirement inequalities are added.

We connect the finite and perpetual interval systems using additional inequalities. Unlike the way intervals were connected in Section 4, the flow through an FSA does not pass from the finite interval to the perpetual interval through a connection variable; the flow through the finite interval can exit via a perpetual variable p_j at any state j that is part of an SCC, and then a cyclic flow (with no beginning or end) is forced to occur in that SCC as part of the perpetual interval. To achieve this, we add a *perpetual-force inequality* $\sum_{k \in out(j)} [x_{2,k}] \geq p_j$ for each state j in an SCC. We use the same subscripting for transition variables as in Section 4: interval 1 is the finite interval, interval 2 is the perpetual interval. This inequality requires that if the FSA containing state j starts repeating transitions perpetually at state j, then there must be a nonzero flow through state j in the perpetual interval.

If the flow through an FSA in the finite interval exits via an accept variable rather than a perpetual variable, then no cyclic flows are forced to occur in that FSA. In fact, in this case we want to prevent such cyclic flows in the FSA. We could enforce this for an FSA N using the quadratic equation

$$\left(\sum_{k \in SCC(N)} [x_{2,k}] \right) \left(1 - \sum_{j \in states(N)} [p_j] \right) = 0$$

where *states(N)* is the set of states of FSA N and $SCC(N)$ is the set of transitions of FSA N contained in an SCC (note that $\sum_{j \in states(N)} [p_j] \leq 1$). In practice, since quadratic programming is much harder than linear programming, we achieve the desired relation using a linear inequality by using a large upper bound U for the transition variables (i.e., $x_{i,k} \leq U$). For each FSA N, we add a *perpetual-bound inequality*,

$$\left(\sum_{k\in SCC(N)} [x_{2,k}]\right) \leq |SCC(N)| \cdot U \left(\sum_{j\in states(N)} [p_j]\right)$$

where $|SCC(N)|$ is the size of set *SCC(N)*. If the flow through the FSA in the finite interval exits via an accept variable (i.e., $\sum_{j\in states(N)}[p_j] = 0$) then no flow is allowed in the FSA in the perpetual interval. If, however, the flow exits via a connection variable (i.e., $\sum_{j\in states(N)}[p_j] = 1$), then this inequality allows any amount of flow (less than U) through arcs in the FSA in the perpetual interval.

The inequalities described are necessary conditions for the existence of a potentially infinite trace. The inequality system representing necessary conditions for the existence of a potentially infinite trace of the example of Figure 1 is shown in Figure 6. We may test for the possibility that M_2 becomes permanently blocked by adding the equation $x_{1,4} = 1$. The resulting inequality system has a solution corresponding to an infinite trace in which transition 4 is taken once and transitions 1 and 5 are taken perpetually ($x_{2,1} = x_{2,5} = 1$). This proves that the b communication need not eventually occur. Most systems would enforce some type of fairness in the selection of a communication partner that would prevent this behavior. We can enforce certain types of fairness using additional inequalities that might, for example, forbid the starvation of an FSA waiting for a communication if that communication is enabled infinitely often, which we can tell from the presence of certain events in the perpetual interval. In the example of Figure 1, the inequality $Ux_{1,4} + x_{2,5} \leq U$ would prevent M_2 from becoming blocked waiting for a communication with M_3 on channel b if M_3 is in state 5 infinitely often. Adding this inequality produces an inequality system with no integral solutions, proving that M_2 cannot become permanently blocked if M_3 must engage in a communication on b that is infinitely often possible. The generation of fairness inequalities is discussed further in [11].

The algorithm to generate inequalities representing necessary conditions for the existence of an infinite trace is shown in Figure 7. The correctness of these conditions is proved in [11]. The technique to find infinite traces can be combined with the technique of Section 4, allowing us to produce necessary conditions for the existence of an infinite trace generated by an ω-star-less expression. To accomplish this, we make the last interval of each sequence a perpetual interval and connect it to the preceding interval just as the perpetual interval was connected to the finite interval above. We add accept variables to the last finite interval, allowing the FSAs to terminate rather than run forever. As a result, the conditions we produce are also necessary for the existence of a finite trace generated by the finite part of the ω-star-less expression (obtained by removing the T_i sets). The size of the inequality system generated by this technique is linear in the size of the automata and linear in the size of the ω-star-less expression.

6. Experiments

The technique for verifying properties expressible with ω-star-less expressions has been implemented in the Inequality Necessary Condition Analyzer (INCA), a descendant of

$$
\begin{array}{rcll}
\textbf{Flow (finite):} & & & \textbf{(state)} \\
1 + x_{1,1} & = & x_{1,1} + x_{1,2} + p_1 & (1) \\
x_{1,2} & = & f_2 & (2) \\
1 + x_{1,3} & = & x_{1,3} + x_{1,4} + p_3 & (3) \\
x_{1,4} & = & f_4 & (4) \\
1 + x_{1,5} & = & x_{1,5} + x_{1,6} + x_{1,9} + p_5 & (5) \\
x_{1,6} + x_{1,7} & = & x_{1,7} + x_{1,8} + p_6 & (6) \\
x_{1,9} & = & x_{1,10} & (7) \\
x_{1,8} + x_{1,10} & = & f_8 & (8) \\
\textbf{Synchronization (finite):} & & & \textbf{(channel)} \\
x_{1,1} & = & x_{1,5} & (a) \\
x_{1,3} & = & x_{1,6} + x_{1,7} & (b) \\
\textbf{Hang (finite):} & & & \textbf{(channel)} \\
x_{1,2} + x_{1,9} & \leq & 1 & (a) \\
x_{1,4} + x_{1,8} + x_{1,10} & \leq & 1 & (b) \\
\textbf{Flow (perpetual):} & & & \textbf{(state)} \\
x_{2,1} & = & x_{2,1} & (1) \\
x_{2,3} & = & x_{2,3} & (3) \\
x_{2,5} & = & x_{2,5} & (5) \\
x_{2,7} & = & x_{2,7} & (6) \\
\textbf{Synchronization (perpetual):} & & & \textbf{(channel)} \\
x_{2,1} & = & x_{2,5} & (a) \\
x_{2,3} & = & x_{2,7} & (b) \\
\textbf{Perpetual-force:} & & & \textbf{(state)} \\
x_{2,1} & \geq & p_1 & (1) \\
x_{2,3} & \geq & p_3 & (3) \\
x_{2,5} & \geq & p_5 & (5) \\
x_{2,7} & \geq & p_6 & (6) \\
\textbf{Perpetual-bound:} & & & \textbf{(FSA)} \\
x_{2,1} & \leq & Up_1 & (1) \\
x_{2,3} & \leq & Up_3 & (2) \\
x_{2,5} + x_{2,7} & \leq & 2U(p_5 + p_6) & (3)
\end{array}
$$

Figure 6. Inequality System for a Potentially Infinite Trace

```
Input:    A set $M$ of FSAs
Output:   A set of inequalities

For each transition $k$ in an FSA of $M$:
    Create transition variable $x_{1,k}$
    If $k$ is in an SCC then
        Create transition variable $x_{2,k}$
For each state $j$ in an SCC of an FSA of $M$:
    Create perpetual variable $p_j$
For each accepting state $j$ of an FSA of $M$:
    Create accept variable $f_j$
For each interval $i = 1, 2$:
    For each state $j$ of an FSA of $M$:
        Generate flow equation:
            $[1]_{i=1 \wedge start(j)} + \sum_{k \in in(j)} x_{i,k} = \sum_{k \in out(j)} x_{i,k} + [p_j]_{i=1} + [f_j]_{i=1}$
For each channel $c$:
    For each interval $i = 1, 2$:
        Generate synchronization equation:
            $\sum_{k \in call(c)} x_{i,k} = \sum_{k \in accept(c)} x_{i,k}$
    Generate restriction inequality:  $\sum_{k \in hang\_c(c)} x_{1,k} + \sum_{k \in hang\_a(c)} x_{1,k} \leq 1$
For each state $j$ in an SCC of an FSA of $M$:
    Generate perpetual-force inequality:  $\sum_{k \in out(j)} [x_{2,k}] \geq p_j$
For each FSA $M_i$ in $M$:
    Generate perpetual-bound inequality:
        $\sum_{k \in SCC(M_i)} [x_{2,k}] \leq |SCC(M_i)| \cdot U \left( \sum_{j \in states(M_i)} [p_j] \right)$
```

Figure 7. Algorithm for Potentially Infinite Trace

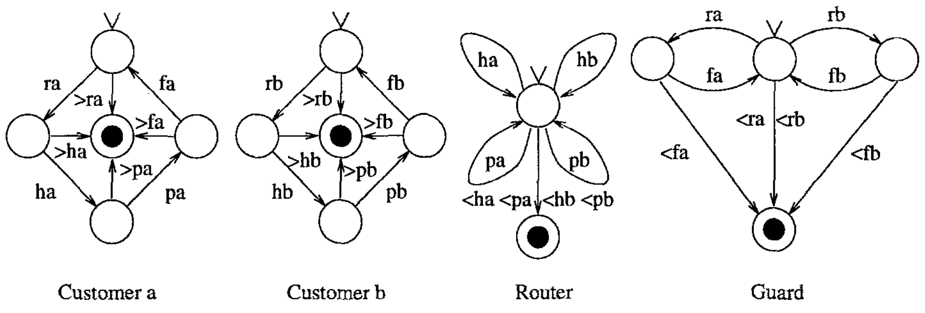


Figure 8. Packet Router Example

the constrained expression toolset [3]. A series of experiments has demonstrated the feasibility of the technique for verifying different kinds of properties on several examples of concurrent systems. This section describes some of these experiments. All times reported are in seconds on a DECstation 5000.

The first experiment we describe is very small and is presented as a demonstration of how an analysis proceeds. The concurrent system shown in Figure 8 contains two customer FSAs (a and b), one router FSA, and one guard FSA. We use the more compact symbols $>c$, $<c$ as the hang symbols for channel c. Customer a (respectively, b) repeats the following forever: communicate with the guard on channel ra (rb) to gain exclusive access to the router, send the header of a packet to the router on channel ha (hb), send the packet to the router on channel pa (pb), and free the router by communicating with the guard on channel fa (fb). The guard guarantees that the router is used in a mutually exclusive fashion. The router simply accepts any packet or header at any time. Present but not shown are restriction languages, like those in the example of Figure 1, that forbid both hang symbols for a channel from occurring in the same trace.

First we verified the safety property that the router cannot send a header for one customer followed immediately by a packet from the other. This can be expressed in linear temporal logic as $\Box[(ha \rightarrow \neg pb\mathcal{U}pa) \wedge (hb \rightarrow \neg pa\mathcal{U}pb)]$. Its negation can be expressed by the star-less expression

$$\Sigma^*\ ha\ (\Sigma - \{pa\})^* pb\ \cup \Sigma^*\ hb\ (\Sigma - \{pb\})^* pa$$

Starting with a specification of the concurrent system in an Ada-like design language and the above expression, the toolset produced an inequality system of 107 inequalities in 128 variables. Our integer programming package determined this inequality system has no integral solution in two seconds. Since the inequality system represents necessary conditions for the existence of a prefix of a trace generated by the expression, we may conclude that the safety property holds.

The second property we attempted to verify was the liveness property that the first customer would transmit a header infinitely often. This property can be expressed in linear temporal logic as $\Box\Diamond ha$ and its negation by the ω-star-less expression $\Sigma^*\ (\Sigma - \{ha\})^\omega$. The toolset produced an inequality system of 67 inequalities in 70 variables and the integer programming package found a solution to this system in one second. Examination of the solution reveals that it does correspond to a possible trace of the concurrent system, one in which customer a becomes permanently blocked waiting to acquire the router while customer b repeatedly acquires it forever. Thus we have proved that the the liveness property does not hold by producing a trace violating the property. The problem is that no fairness is enforced when selecting a communication partner. When we instruct the toolset to produce two additional inequalities to enforce fairness in the guard's selection of a communication partner, as described in Section 5, the resulting inequality system was determined to have no integral solution in three seconds. This proves that the liveness property does hold, assuming an FSA cannot starve waiting for a communication that is infinitely often possible.

In the absence of such fairness, it is possible to verify a weaker liveness property: once a customer (say a) has acquired access to the router, it must eventually get to transmit a packet. This can be expressed in linear temporal logic by the formula $\Box(ra \rightarrow \Diamond ha)$ and its negation by the ω-star-less expression $\Sigma^* \ ra(\Sigma - \{ha\})^\omega$. The toolset produced an inequality system of 59 inequalities in 56 variables which was found to have no integral solution in one second, proving this weaker liveness property holds even in the absence of fairness.

The second experiment we describe involves several versions of a mutual exclusion protocol based on the concept of *coteries* [12]. A coterie is a general mechanism for achieving mutually exclusive access to a resource in a distributed system. Each resource has a set of keys. A coterie is a set of subsets of the keys with the property that any two subsets have a non-null intersection. A customer wishing to use the resource must first acquire all of the keys in one of the coterie subsets. Since any two coterie subsets must share at least one key, only one customer may possess all of the keys in one of the coterie subsets at any given time. One simple type of coterie consists of all subsets possessing a majority of the keys.

We considered four versions of a coterie mutual exclusion system containing one resource, three guards, and either two or three customers. The customers must acquire a majority of the keys (i.e., two keys) to use the resource. Each guard holds one key and a customer communicates with a guard to request, acquire, and release its key. In the first two versions, there are three customers which each request two specific keys, waiting until these keys are granted before using the resource; in the second two versions, there are two customers which cyclically request each of the three keys until they have been granted a total of two keys. In the second and fourth versions, the guards are extended to enforce fairness in granting the requests: In the second version, the guards queue requests for the keys; in the fourth version, a guard will not grant a key to the customer that last held it if the other customer has since requested the key.

Two properties of mutual exclusion protocols are commonly verified. First, the protocol must enforce the mutual exclusion. Second, the protocol should be "fair" in some sense—a customer wishing to use the resource should eventually be permitted to do so. The first property is a safety property; the second is (usually) a liveness property. We attempted to verify four properties: freedom from deadlock, mutual exclusion, freedom from starvation (this is the fairness property mentioned above), and a queuing property for the second version (essentially a strong kind of fairness that is actually a safety property). The performance of the automated tools on these examples is shown in Table 2. It lists, for each example and property: the result of the experiment, the time to generate the inequality system from an Ada-like specification, the time to solve the system, the total time, and the size of the inequality system generated. We also analyzed incorrect versions of the systems to show that the necessary conditions generated are useful for finding errors when they are present; incorrect versions have an "(i)" after the version number. The suffix "-fair" on the starve properties in the first version indicate that fairness inequalities were generated to enforce fairness in selecting communication partners.

There are four possible results of an experiment: "verified" indicates that the ILP package determined that the inequality system generated had no integral solutions, proving the property holds; "disproved" indicates that the ILP package found a solution that corresponds to a trace violating the property, thus proving that the property does not hold; "ILP fails" indicates that the ILP package could not determine (within a preset bound on the number of branch and bound iterations) whether or not the inequality system has an integral solution; "spurious" indicates that an integral solution to the inequality system was found but this solution did not correspond to a trace of the system. In the last two cases, the analysis is inconclusive. Each of these cases happened once in this set of experiments. Each time, however, we were able to simplify the property by breaking it down into cases and verify each case (the suffix "-1" idicates that the property is one case of the full property).

In summary, the modified toolset was able to give a definitive answer for all the properties we attempted to verify. Although these systems do not have a large number of processes, the processes themselves are fairly complicated (some having over 200 states in their FSA representation) and together the system they comprise is complex enough to possess a variety of interesting properties. As Table 2 shows, the average analysis time to verify or disprove a property of these systems, starting from an Ada-like source and an ω-star-less expression, was about five minutes. These experiments are described in great detail in [11].

The last experiment described here involves a version of the dining philosophers system where the standard deadlock is prevented by having the philosophers pass around a "dictionary"; the philosopher holding the dictionary cannot hold any forks. Each philosopher nondeterministically decides whether to read or eat. If she decides to read, she waits for the philosopher to her right to pass her the dictionary, reads it, then waits for the philosopher to her left to accept the dictionary. If she decides to eat, she picks up her left fork, then her right fork, eats, and puts down her forks. This activity is repeated forever. Each philosopher and fork is modeled with a process. The philosopher processes synchronize with each other to pass the dictionary and synchronize with the fork processes to pick up and put down the forks. In this example, we verify the same properties on several sizes of the system to get an idea of how the techniques perform as the size of the system is scaled up.

We attempted to verify several properties for each size of the system:

Table 2. Toolset Performance on Coterie Mutual Exclusion

				Time		Size
Version	Property	Result	Gen	Solve	Total	Ineqs×Vars
1	mutex	verified	73	258	331	234×272
1 (i)	mutex	disproved	68	4	72	220×254
1	deadlock	verified	51	1	52	93×102
1	starve	disproved	57	14	71	154×150
1	starve-fair	verified	57	19	76	172×150
1 (i)	starve-fair	disproved	72	3	75	170×147
2	mutex	verified	364	1140	1504	365×414
2 (i)	mutex	disproved	370	8	378	356×405
2	queue	verified	254	2	256	315×410
2 (i)	queue	disproved	262	3	265	319×419
3	mutex	ILP fails	329	2483	2812	591×661
3	mutex-1	verified	280	4	284	381×537
3 (i)	mutex	disproved	327	36	363	570×638
3	starve	disproved	291	34	325	467×511
4	mutex-1	verified	360	7	367	552×855
4 (i)	mutex-1	disproved	352	5	357	449×668
4	starve	spurious	381	74	455	650×742
4	starve-1	verified	369	10	379	549×598
4 (i)	starve-1	disproved	352	25	377	457×484
4	deadlock	verified	324	317	641	231×334

1. Philosopher 0 cannot become permanently blocked waiting to pick up her left fork. This property holds, even without the dictionary.

2. Philosopher 0 cannot become permanently blocked waiting to pick up her right fork. Without the dictionary, this property does not hold since all of the philosophers may become permanently blocked waiting to pick up their right forks. With the dictionary, however, the property holds, proving that the deadlock is prevented.

3. Philosopher 0 cannot become permanently blocked waiting to get the dictionary. Since some philosopher between philosopher 0 and the philosopher currently holding the dictionary may never decide to read again, this property does not hold.

4. Philosopher 0 cannot become permanently blocked waiting to pass the dictionary to the next philosopher. Since Philosopher 1 may never decide to read again, this property does not hold.

The results of these analyses are shown in Table 3. In the first two cases, the toolset determined correctly that the property holds. In the latter two cases, the toolset found a solution corresponding to an execution of the system in which the property is violated.

The last two properties do not hold because a philosopher may stop reading after some finite time. We therefore attempted to verify them under the assumption that each philosopher reads infinitely often. Results of these analyses are shown in Table 4. In both cases, the toolset correctly determined that the properties hold under the fairness assumption. Note that though the size of the state space of this system is growing exponentially in the number of philosophers, our analysis times appear to be growing much more slowly.

Table 3. Toolset Performance on Dining Philosophers with Dictionary

	Number			Time		Size
Property	of Phils	Result	Gen	Solve	Total	Ineqs×Vars
left-fork	20	verified	214	14	228	1035×994
	40	verified	508	121	629	2075×1994
	60	verified	800	176	976	3115×2994
	80	verified	1305	276	1581	4155×3994
	100	verified	1736	371	2107	5195×4994
right-fork	20	verified	216	17	233	1036×995
	40	verified	481	44	525	2076×1995
	60	verified	846	99	945	3116×2995
	80	verified	1304	113	1417	4156×3995
	100	verified	1756	179	1935	5196×4995
get-dict	20	disproved	210	152	362	958×997
	40	disproved	487	325	812	1918×1997
	60	disproved	828	488	1316	2878×2997
	80	disproved	1245	842	2087	3838×3997
	100	disproved	1692	1413	3105	4798×4997
dump-dict	20	disproved	230	43	273	958×996
	40	disproved	467	133	600	1918×1996
	60	disproved	836	268	1104	2878×2996
	80	disproved	1245	436	1681	3838×3996
	100	disproved	1684	652	2336	4798×4996

Table 4. Toolset Performance on Dining Philosophers with Dictionary When Philosophers Must Read Infinitely Often

	Number			Time		Size
Property	of Phils	Result	Gen	Solve	Total	Ineqs×Vars
get-dict	20	verified	215	22	237	983×1024
	40	verified	500	55	555	1963×2044
	60	verified	870	144	1014	2943×3064
	80	verified	1350	244	1594	3923×4084
	100	verified	1894	390	2284	4903×5104
dump-dict	20	verified	212	16	228	983×1024
	40	verified	528	61	589	1963×2044
	60	verified	876	137	1013	2943×3064
	80	verified	1336	214	1550	3923×4084
	100	verified	1879	354	2233	4903×5104

Finally, we mention that we also attempted to verify the last two properties under the weaker fairness assumption that a philosopher who does not become permanently blocked will choose to read infinitely often. The toolset determined that the properties hold. For the last property, the times were comparable to those reported in the table. In testing whether a philosopher could become blocked waiting to get the dictionary, however, we encountered numerical problems in solving the inequalities representing the dining philosophers systems with this fairness assumption. The times required to solve the system of inequalities were significantly larger in this case, and increased more rapidly with the number of philosophers. For instance, it took approximately 8000 seconds to solve the system with 60 philosophers and more than 7 hours to solve the system with 100 philosophers.

For comparison, we note that the BDD-based technique of [15] can verify similar properties of these systems up to 48 philosophers, but was unable to handle a 64 philosopher system.

7. Conclusion

We have presented a technique for verifying many safety and liveness properties of concurrent systems. The technique involves generating linear inequalities that represent necessary conditions for the existence of a trace violating the property. The obvious advantage of the approach is that it does not require enumeration of the system's many states. The disadvantages are that spurious solutions to the inequality system can make the analysis inconclusive and the tractability of integer linear programming in practice is not well understood. Nevertheless, our experience [3] suggests that spurious solutions are relatively rare and that our inequality systems, being largely network flow systems, have a special structure that usually makes their solution tractable. Furthermore, a prototype implementation of the technique has been successfully applied to several sample systems, including systems with relatively few complex processes and systems with many simple processes. In essence, our technique sacrifices some generality for tractability. If our necessary conditions do not sufficiently restrict the order of interprocess interactions, there may be spurious solutions to our inequalities, making our analysis inconclusive. For many systems and properties, however, our necessary conditions are strong enough to yield appropriate orderings of interprocess interactions, and the method is both tractable and conclusive.

The practical utility of our technique depends on the strength of its necessary conditions and the tractability of the integer programming problems it produces. Although it would be desirable to have a formal characterization of the class of systems for which our technique is practical, we believe that the best information about the practicality of our method will come from accumulating experience in applying it to a wide range of systems. The most important reason for this is that sufficient information about the tractability of the integer programming problems is not available. Although there has been some effort to develop special-purpose solvers for this particular type of problem, network flows with certain side constraints, we are not aware of any theoretical results that indicate precisely when such problems can be solved much more easily than the general integer linear programming problem. Even papers proposing new methods for solving such problems validate those methods by presenting empirical data on their performance on standard test problems (e.g., [1]). We therefore plan a major project involving a thorough empirical evaluation of our technique, including both its application to a large number of sample concurrent systems and comparison with other techniques such as symbolic model checking.

Acknowledgments

The research described here was partially supported by National Science Foundation grant CCR-9106645 and Office of Naval Research grant N00014-89-J-1064. Many thanks to David Mix-Barrington for clarifying the expressibility of star-less expressions.

Notes

1. Not to be confused with a *star-free* regular expression, to be discussed below. We use the term *star-less* since the expressions specify patterns of the $e_{i,j}$ events using only concatenation and union (allowing the intervening symbols specified by the $S_{i,j}$).

References

1. A. I. Ali, J. Kennington, and B. Shetty. The equal flow problem. *European J. Oper. Res.*, 36:107–115, 1988.
2. G. S. Avrunin, U. A. Buy, and J. C. Corbett. Integer programming in the analysis of concurrent systems. In Larsen and Skou [16], pages 92–102.
3. G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, 17(11):1204–1222, Nov. 1991.
4. J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Comput. Sci.*, 37(1):77–121, 1985.
5. R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.
6. J. Burch, E. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.
7. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.
8. E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, Jan. 1992.
9. E. M. Clarke and R. P. Kurshan, editors. *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, Providence, RI, 1991. American Mathematical Society.
10. R. Cleaveland, J. Parrow, and B. Steffen. The concurrency workbench: A semantics based tool for the verification of concurrent systems. *ACM Trans. Prog. Lang. Syst.*, 15(1):36–72, Jan. 1993.
11. J. C. Corbett. *Automated Formal Analysis Methods for Concurrent and Real-Time Software*. PhD thesis, University of Massachusetts at Amherst, 1992.
12. H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *J. ACM*, 32(4):841–860, Oct. 1985.

13. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Larsen and Skou [16], pages 332–242.
14. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.
15. R. Hojati, H. Touati, R. P. Kurshan, and R. K. Brayton. Efficient ω-regular language containment. In G. v. Bochmann and D. K. Probst, editors, *Computer Aided Verification, 4th International Workshop Proceedings*, volume 663 of *Lecture Notes in Computer Science*, pages 371–382, Montreal, Canada, 1992. Springer-Verlag.
16. K. G. Larsen and A. Skou, editors. *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, Aalborg, Denmark, July 1991. Springer-Verlag.
17. R. Milner. *Communication and Concurrency.* Prentice Hall, London, 1989.
18. D. K. Probst and H. F. Li. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In Clarke and Kurshan [9], pages 15–24. Also LNCS 531, pp. 15–24.
19. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. MIT Press/Elsevier, 1990.
20. A. Valmari. Compositional state space generation. In *European Conference on Petri Nets*, pages 43–62, 1990.
21. A. Valmari. A stubborn attack on state explosion. In Clarke and Kurshan [9], pages 25–41.
22. W. J. Yeh and M. Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, Oct. 1991. ACM SIGSOFT, Association for Computing Machinery.
23. H. Zuidweg. Verification by abstraction and bisimulation. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 105–166, June 1989. Appeared as *Lecture Notes in Computer Science* 407.