# Verifying General Safety and Liveness Properties With Integer Programming*

James C. Corbett

Information and Computer Science Department
University of Hawaii at Manoa

**Abstract.** Analysis of concurrent systems is plagued by the state explosion problem. The *constrained expression* analysis technique 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. We extend this 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.

## 1 Introduction

Many concurrent systems can be modeled as a set of communicating finite state machines. Analysis of such systems is generally difficult, however, since the number of system states grows exponentially with the number of state machines. Many techniques have been proposed to cope with this state explosion problem, including symbolic model checking [3], partial order techniques [6, 10], and compositional techniques [4]. Last year at the Third Workshop on Computer Aided Verification, another technique was presented [1] that involves the use of necessary conditions to answer certain types of questions about a system without enumerating the system's states. The technique has been automated as part of the *constrained expression toolset* and has been applied to some concurrent systems having as many as $10^{47}$ reachable states [2]. Unfortunately, the types of questions that can be answered by this technique are somewhat limited. For example, it can determine if a system could deadlock, but it cannot address liveness questions, which involve infinite traces, nor can it directly address questions like mutual exclusion, which involve the relative order of events. This paper extends the technique to handle both infinite traces and questions about the relative order of events. A further extension of these ideas in [5] 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.

* The research described here was partially supported by National Science Foundation grant CCR-9106645 and Office of Naval Research grant N00014-89-J-1064.

## 2 Model and Basic Technique

As in [1], 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, a concurrent system is 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 $A_1, \ldots, A_m$, where $A_i \subseteq \Sigma$ 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 [1], but in this paper we will focus on the case where pairs of processes communicate synchronously over named channels that connect them. We model such a communication using the channel name as an event symbol that appears in the alphabets of the FSAs of both processes. The possibility that the task becomes permanently blocked waiting for the communication is represented by the choice to accept a *hang symbol* for that channel rather than engage in the communication. The hang symbols for channel $a$ are denoted $>a$ and $<a$ (there is one hang symbol for each of the two tasks connected by the channel). For each channel $a$, the restriction language $\{>a, <a, \lambda\}$ forbids hang symbols from both ends of the channel from occurring. A small example is shown in Fig. 1. In all our examples, we shall take the set of event symbols appearing in an FSA or restriction language as its alphabet.

The basic technique, detailed in [1], 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. A trace can be viewed as a path in each FSA from the starting state to an accepting state such that the interactions between the FSAs represented by the paths are consistent. Our technique finds a flow in each FSA from the starting state to an accepting state such that the flows satisfy a weaker consistency criterion. Specifically, we require that for each communication channel, the FSAs connected by that channel agree on the number of times that they communicated over that channel.

To produce the inequalities, we assign a variable, $x_i$, called a *transition variable*, to each transition $i$ in the FSAs 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. We then produce a *flow equation* for each state,

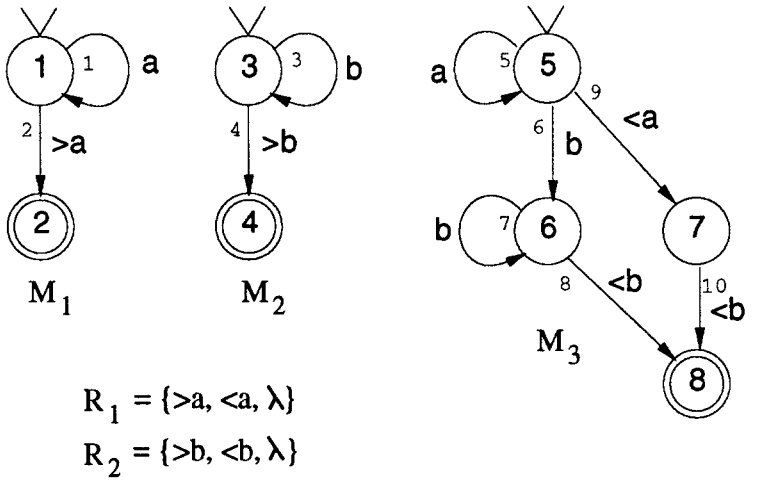


**Fig. 1.** Small example

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 in of one at the start state and accept variables are counted as flow out. We also produce a *communication equation* for each channel, equating the number of times the processes connected by the channel communicated over that channel. Finally, we 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 inequalities produced for the example of Fig. 1 are shown in Fig. 2.

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, however, not every set of flows satisfying the communication and restriction inequalities will correspond to a trace. There are two 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 A and then channel B, while the other communicated over channel B and then channel A). Secondly, the presence of cycles in the FSAs can allow cyclic flows that 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 Fig. 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. 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

| | |
|---|---|
| **Flow:** | **(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) |
| **Communication:** | **(channel)** |
| $x_1 = x_5$ | ($a$) |
| $x_3 = x_6 + x_7$ | ($b$) |
| **Restriction:** | **(number)** |
| $x_2 + x_9 \leq 1$ | (1) |
| $x_4 + x_8 + x_{10} \leq 1$ | (2) |

**Fig. 2.** Inequality System for Finite Trace

another solution that does correspond to a trace. In our experience [2], however, such spurious solutions are uncommon. Also, we can sometimes add additional inequalities to remove such solutions.

Similar techniques have been used to prove structural properties of Petri nets (e.g., boundedness, repetitiveness) using transition matrices. Space limitations preclude a detailed comparison with this work, which is reviewed in [8].

## 3 Extended Techniques

In this paper, we extend the basic technique presented in the last section to the verification of properties specified by an $\omega$-regular expression [9] of the form:

$$\bigcup_{i=1}^{m} S_{i,0}^* e_{i,1} S_{i,1}^* e_{i,2} \ldots S_{i,n_i-1}^* e_{i,n_i} S_{i,n_i}^* T_i^\omega$$

where $S_{i,j} \subseteq \Sigma$, $e_{i,j} \in \Sigma$, $T_i \subseteq \Sigma$. We call such an expression an *$\omega$-star-less* expression[2]. Specifically, given an $\omega$-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 Sect. 4. The second idea allows the technique to deal with infinite traces and is described in Sect. 5.

[2] Not to be confused with *star-free* expressions. We call these expressions star-less since they specify patterns of the $e_{i,j}$ events using only concatenation and union (allowing the intervening symbols specified by the $S_{i,j}$). Star-free expressions allow concatenation, union, and negation, but not Kleene star.

In [5], these ideas are carried further to allow the verification of properties specified by a Büchi automaton. It is well known that Büchi automata are more expressive than first order logic [9]. This implies that this further extension suffices to handle any property expressible in linear temporal logic. We do not present this further extension here for two reasons. First, although it relies on the same two ideas, it is significantly more complicated to describe. Second, unlike the first extension, it has not been implemented and tried on sample systems; hence the quality of the necessary conditions it produces, and thus its practical significance, is not known.

Using the extended technique presented here, we can verify that a system has any property whose negation is expressible as an $\omega$-star-less expression. To accomplish this, we use the extended technique to produce necessary conditions, in the form of linear inequalities, for the existence of a trace of the system generated by the $\omega$-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 (i.e., a solution to the inequality system usually corresponds to a trace violating the property). 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

The technique presented in Sect. 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 Fig. 1, we would add $x_1 = 1$ and $x_3 = 1$ to the inequality system in Fig. 2. 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 conceptually divide the trace into *intervals* using those events, produce a different inequality system for each interval, and connect these inequality systems together.

We will explain the technique using the example of Fig. 1. 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 $\omega$-star-less expression $\Sigma^* b(\Sigma - \{a, b\})^* a\Sigma^\omega$. We produce necessary conditions for the existence of a prefix of a trace containing a $b$ followed by an $a$, as generated by the finite part of the expression (since the above is a safety property, the infinite suffix after the violation, generated by $\Sigma^\omega$, can be ignored). 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 in Fig. 2, 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 $i$ having an incoming $b$ transition a *connection variable* $c_{1,i}$ that will be one if the FSA containing state $i$ is in state $i$ 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 equation* 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, rather than having accept variables, 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 the FSA could be in after an $a$ event. We assign connection variables $c_{2,i}$, representing the number of times the second interval ends at state $i$, to states that the FSAs could be in following an $a$ event. 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, rather than having accept variables. Finally, the restriction inequalities are produced as before and involve the number of hang symbols from both intervals.

The inequality system produced for this example is shown in Fig. 3. 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 a prefix of a trace containing a $b$ and then an $a$ to exist, has no integral solution. This proves that no trace generated by the expression $\Sigma^* b(\Sigma - \{a, b\})^* a\Sigma^\omega$ 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 will work even if the events $a$ and $b$ are in different FSAs, as shown by an example in Sect. 6.

We have shown how to produce necessary conditions for the existence of a trace containing a specific sequence of events. We can produce necessary conditions for a trace generated by the union of such sequences as follows. We assign a

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

**Fig. 3.** Inequality System for Prefix of Trace Generated by $\Sigma^* b(\Sigma - \{a, b\})^* a$

*sequence variable* $s_i$ to each sequence that will be one if that sequence is the one found and zero otherwise. We produce an equation summing the sequence variables to one, forcing one sequence to be sought. We produce inequality systems for each sequence as described above and connect them as follows. The implicit flow into the start states of each FSA in the first interval of sequence $i$ is set to $s_i$ rather than to one, thus flows will only be found in the inequality system for one sequence. Also, the requirement equations are changed to require that the events ending the intervals of that sequence occur $s_i$ times (we cannot force an event in

a sequence to occur unless the sequence occurs). The resulting inequality system represents necessary conditions for the existence of a trace generated by one of the sequences.

## 5 Infinite Traces

Another limitation of the technique presented in Sect. 2 is that it does not admit infinite traces, i.e., traces in which one or more FSAs continue engaging in actions forever. Note that the inequality system in Fig. 2 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). To test for liveness properties, we must be able to represent infinite traces since the negation of a liveness property will be an expression forbidding some good event(s) from occurring in a potentially infinite execution.

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. We use the term interval in the same technical sense as in the last section: each interval has its own transition variables and inequalities. The perpetual interval, however, represents an infinite suffix of a trace using a finite string of the events that are repeated forever in the suffix (this representation loses information about the order in which these events are repeated). The occurrence of an event in this interval represents the event being repeated infinitely often in the trace.

We produce inequalities for the two intervals as we did in the last section, but with the following differences. In each FSA, the set of transitions taken in the perpetual interval must form a strongly connected component (SCC) of the FSA when viewed as a graph[3]. Therefore, when generating inequalities for the perpetual interval, we include only transitions that are part of SCCs (in the example of Fig. 1, this consists of transitions 1, 3, 5, and 7). We assign connection variables to all states that are part of SCCs, allowing the finite part of the FSA's behavior to end at any point at which it could start repeating events. We then add additional *perpetual inequalities* to force a cyclic flow (a flow with no beginning or end) to occur in an SCC of the perpetual interval if that interval is "entered" via a connection variable. Unlike the case described in Sect. 4, the flow through the FSA does not pass from one interval to another through the connection variable; the flow through the finite interval simply ends at some state in the FSA that is part of an SCC and we then force a cyclic flow to occur in the SCC as part of the perpetual interval. The flow equations for the perpetual interval do not contain connection or accept variables; the only possible flows are cyclic. For each state $j$, let $P_j$ be the set of transitions out of

[3] Strictly speaking, SCCs are composed of nodes, not arcs. We say that an arc is part of an SCC if there exists some SCC containing both of the nodes connected by the arc.

$j$ that are part of an SCC. For each state $j$ where $P_j \neq \emptyset$, we add a perpetual inequality $\Sigma_{i \in P_j} x_{2,i} \geq c_{1,j}$. This inequality requires that if the FSA containing state $j$ enters the perpetual interval at state $j$, then there must be a cyclic flow through state $j$ in the perpetual interval. Of course, a particular FSA may not run forever, even in an infinite trace. Accept variables allow the flow through an FSA in the finite interval to stop without forcing the occurrence of events in the perpetual interval.

| | |
|---|---|
| **Flow (finite):** | **(state)** |
| $1 + x_{1,1} = x_{1,1} + x_{1,2} + c_{1,1}$ | (1) |
| $x_{1,2} = f_2$ | (2) |
| $1 + x_{1,3} = x_{1,3} + x_{1,4} + c_{1,3}$ | (3) |
| $x_{1,4} = f_4$ | (4) |
| $1 + x_{1,5} = x_{1,5} + x_{1,6} + x_{1,9} + c_{1,5}$ | (5) |
| $x_{1,6} + x_{1,7} = x_{1,7} + x_{1,8} + c_{1,6}$ | (6) |
| $x_{1,9} = x_{1,10}$ | (7) |
| $x_{1,8} + x_{1,10} = f_8$ | (8) |
| **Communication (finite):** | **(channel)** |
| $x_{1,1} = x_{1,5}$ | ($a$) |
| $x_{1,3} = x_{1,6} + x_{1,7}$ | ($b$) |
| **Flow (perpetual):** | **(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) |
| **Communication (perpetual):** | **(channel)** |
| $x_{2,1} = x_{2,5}$ | ($a$) |
| $x_{2,3} = x_{2,7}$ | ($b$) |
| **Restriction:** | **(number)** |
| $x_{1,2} + x_{1,9} \leq 1$ | (1) |
| $x_{1,4} + x_{1,8} + x_{1,10} \leq 1$ | (2) |
| **Perpetual:** | **(state)** |
| $x_{2,1} \geq c_{1,1}$ | (1) |
| $x_{2,3} \geq c_{1,3}$ | (3) |
| $x_{2,5} \geq c_{1,5}$ | (5) |
| $x_{2,7} \geq c_{1,6}$ | (6) |

**Fig. 4.** Inequality System for Potentially Infinite Trace

The inequalities described comprise necessary conditions for the existence of a potentially infinite trace. The inequality system for the example of Fig. 1 is shown in Fig. 4. We may test for the possible starvation of $M_2$ 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 ($x_{1,4} = x_{2,1} = x_{2,5} = 1$). This tells us that, in the absence of any fairness properties for selection of communication partners, it is not the case that a $b$ communication must eventually occur. We can enforce certain types of fairness using additional inequalities that might, for example, forbid the starvation of an FSA waiting for a communication (e.g., $b$) if that communication is enabled infinitely often, which we can tell from the presence of certain events in the perpetual interval (e.g., the $a$ on transition 5, which indicates that $M_3$ is infinitely often in state 5 in which a transition on $b$ is enabled).

This technique to represent infinite traces can be combined with the technique of Sect. 4, allowing us to produce necessary conditions for the existence of an infinite trace generated by an $\omega$-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. The size of the inequality system generated by these techniques is linear in the size of the automata and linear in the size of the $\omega$-star-less expression. Finally, we note that the conditions produced are also necessary for the existence of a finite trace generated by the finite version of the $\omega$-star-less expression (obtained by replacing all occurrences of $\omega$ with Kleene star).

## 6 Example

The technique described above has been implemented as an extension of the constrained expression toolset [2]. A series of experiments has demonstrated the feasibility of the technique for verifying different kinds of properties on several examples of concurrent systems. In this section, we describe one of the smallest examples and the properties we verified using the technique.

The concurrent system shown in Fig. 5 contains two customer FSAs ($a$ and $b$), one router FSA, and one guard FSA. Customer $a$ ($b$) 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 Fig. 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 $\Box[(ha \rightarrow \neg pb\,U\,pa) \wedge (hb \rightarrow \neg pa\,U\,pb)]$. Its negation can be expressed by the $\omega$-star-less expression

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

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

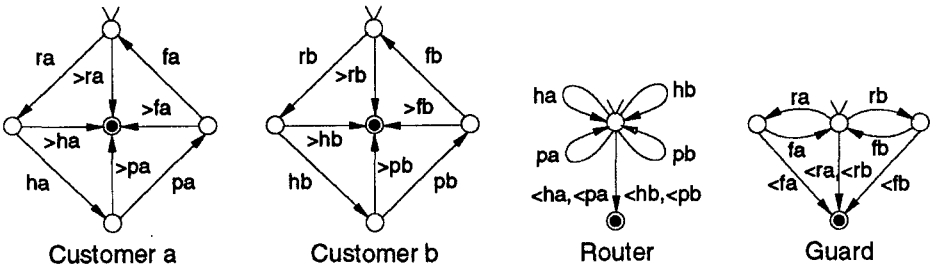


**Fig. 5.** Packet Router Example

107 inequalities in 128 variables. Our integer programming package determined this inequality system has no integral solution in two seconds on our DECstation 5000. Since the inequality system represents necessary conditions for the existence 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 $\omega$-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 Sect. 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 $\omega$-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.

# 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 a trace violating the property to exist. The obvious advantage of the approach is that it does not require enumeration of all possible system 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 [2] 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 demonstrated its feasibility on a range of sample systems [5]. Further experiments in which problem sizes are scaled up, such as those performed in [2] for the original technique, are needed to assess the practicality of this new technique.

# Acknowledgements

This work was done as part of the constrained expression project at the University of Massachusetts directed by George Avrunin and Jack Wileden. Special thanks are due to George Avrunin for suggesting the idea of queries involving the order of events and ideas for its solution.

# References

1. G. S. Avrunin, U. A. Buy, and J. C. Corbett. Integer programming in the analysis of concurrent systems. In Larsen and Skou [7], pages 92–102.
2. 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.
3. 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.
4. E. Clarke, D. Long, and K. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.
5. J. C. Corbett. *Automated Formal Analysis Methods for Concurrent and Real-Time Software*. PhD thesis, University of Massachusetts at Amherst, 1992.
6. P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Larsen and Skou [7], pages 332–242.
7. 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.
8. T. Murata. Petri nets: Properties, analysis and applications. *Proceedings of the IEEE*, 77(4):541–580, Apr. 1989.

9. W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. MIT Press/Elsevier, 1990.
10. A. Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 25–41, Providence, RI, 1991. American Mathematical Society.