Integer Programming in the Analysis of Concurrent Systems[1]

George S. Avrunin
University of Massachusetts
at Amherst
Amherst, MA 01003
avrunin@math.umass.edu

Ugo A. Buy
University of Illinois
at Chicago
Chicago, IL 60680
buy@figaro.eecs.uic.edu

James C. Corbett
University of Massachusetts
at Amherst
Amherst, MA 01003
corbett@cs.umass.edu

1 Introduction

Large computer systems are frequently organized as collections of cooperating asynchronous processes. Their size and the presence of nondeterminacy make it extremely difficult to understand and predict the behavior of such systems. Rigorous analysis methods with automated support will therefore be necessary for the production of reliable software. In this paper, we describe a method for generating and solving systems of inequalities that can be used effectively in the analysis of large computer systems. We also briefly discuss our experience in using this approach in the analysis of concurrent system designs written in an Ada-based design language.

Underlying our approach to analysis is a model in which each asynchronous process in such a system is represented by a finite state automaton (FSA) accepting a language over an alphabet of symbols corresponding to events occurring in executions of that process. An execution of the concurrent system involves executions of each of its component processes, subject to additional restrictions imposed by the appropriate concurrency and communication primitives. These additional restrictions are represented by a set of recursive languages over alphabets contained in the union of the alphabets of the FSAs. A string over the union of the FSA alphabets corresponds to the trace of an execution of the concurrent system if its projection on the alphabet of each FSA lies in the language accepted by that automaton, so the string represents the trace of an execution of the corresponding process, and if its projection on the alphabet of each of the additional recursive languages lies in that language, so the string satisfies the additional restrictions imposed by the concurrency and communication primitives. (For simplicity, in this paper we will regard the executions of concurrent systems as represented by traces, and so as totally ordered in time. Our model and our analysis techniques are, however, compatible with viewing the executions as corresponding only to partial orders of events.)

[1]The research described here was partially supported by National Science Foundation grant CCR-8806970 and Office of Naval Research grant N00014-89-J-1064.

An example of this type of model is the constrained expression formalism [8, 14, 15], which has been used to represent concurrent systems given by Petri nets and by programs in a variety of programming and design languages, involving both synchronous and asynchronous communication [3, 6, 9].

Our approach to analysis is based on the following paradigm. First, we generate a set of linear equations for each FSA. Second, additional linear inequalities are generated that reflect the additional restrictions on the behavior of the model imposed by the recursive languages. Third, inequalities representing the analyst's assumptions or queries are added to the system. Finally, we apply integer programming (IP) techniques to solve the resulting system of equalities and inequalities. If no integer solutions to the linear system exist, we conclude that our conditions are inconsistent and that no trace satisfies the restrictions imposed by the analyst's assumptions. If an integer solution is found, it may or may not correspond to the trace of an actual execution. In this case, however, the solution can be used to guide a highly constrained reachability-based analysis of the concurrent system under consideration.

This approach to analysis has several advantages. It can be applied to the analysis of system descriptions written in a variety of specification and implementation languages. By contrast with reachability-based approaches, this approach does not require the enumeration of the complete state space for the system being analyzed. Finally, the approach has been implemented in the *constrained expression toolset*, a set of prototypes that carry out the analysis of concurrent system designs written in an Ada-based design language. Performance results have been very encouraging and compare favorably with other approaches to analysis [3].

This paper is organized as follows. The model underlying our approach is described in section 2. The analysis method we use is discussed in section 3. The toolset implementing constrained expression analysis of designs is described in section 4 along with some performance results. The last section discusses some related techniques as well as ongoing and planned extensions of our method.

2 The Model

We require the following definitions. For any sets of symbols Σ and S with $S \subseteq \Sigma$, let $\rho_A : \Sigma^* \rightarrow S^*$ be the homomorphism, called *projection on* S, defined by extending the map $\Sigma \rightarrow S$ given by:

$$\rho_S(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in S \\ \lambda & \text{otherwise} \end{cases}$$

Let $L(M)$ be the language recognized by machine M. The *shuffle* of the languages L_1 and L_2, written $L_1 \otimes L_2$, is the language consisting of the strings $x_1 y_1 x_2 y_2 \ldots x_n y_n$ formed by concatenating substrings such that $x_1 x_2 \ldots x_n \in L_1$ and $y_1 y_2 \ldots y_n \in L_2$ for some n (the substrings x_1 and y_n may be empty). Finally, let *dagger* ($\dagger$) be the closure of the shuffle operation: $L^\dagger = \{w | \text{for some } n\text{: } w = w_1 \otimes w_2 \otimes \ldots \otimes w_n, w_i \in L \text{ for all } i\}$.

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 $\Sigma_1, \ldots, \Sigma_n$, 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 $T \subseteq \Sigma = \bigcup_i \Sigma_i$ is a terminal alphabet. 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. An example of the use of this model to describe a system of two processes communicating by asynchronous message passing is given in the next section.

This formulation is a somewhat more general version of the formal definition of *constrained expressions* given in [3]. That formulation allows as restriction languages only those generated by the standard regular operators (concatenation, union, and Kleene star) plus shuffle and dagger. It follows from [1] that all recursively enumerable languages are given by constrained expressions. Allowing arbitrary recursive languages as restrictions, as in our model, thus represents no increase in power, although it may make application of the model somewhat simpler and more convenient.

3 The Method

Given a concurrent program represented in the above model, we generate a system of linear inequalities reflecting much, but not all, of the semantics of the representation to determine if any executions of the concurrent program exist that satisfy certain properties. Essentially, the method finds a possible execution of the concurrent program by finding traces of each process and then enforcing a weaker consistency criterion between these traces than is specified in the restriction languages.

When generating equations for the FSAs $M_1, \ldots, M_n$, it is useful to picture each FSA as a directed graph in the standard way. An execution of the concurrent program will correspond to a path through each FSA from the start state to a final state. We assign a *transition variable* x_a to each transition arc a in the FSAs of the concurrent program. The variable associated with an arc will represent the number of times that arc is crossed in the paths. We also assign an *accept variable* f_i to each final state i of the FSAs that will be one if and only if the path through that FSA ends at this final state and will be zero otherwise. We then generate a *flow equation* for each state i in an FSA stating that the flow into the state (i.e., the total number of times paths enter the state) must equal the flow out of the state (i.e., the total number of times paths leave the state). The start state of each FSA has an implicit flow in of one, and each final state has an extra flow out represented by its accept variable. The flow equations imply that, in any nonnegative integer solution, exactly one accept variable in each FSA will have the value one.

Suppose that a symbol α belongs to two alphabets Σ_i and Σ_j. Then an occurrence of α in a string corresponding to an execution represents the occurrence of an event in the processes corresponding to M_i and M_j. In such a case, we must add an equation stating that the numbers of occurrences of α in the traces of those processes is the same. In other words, we must equate the sum of the transition variables for arcs of M_i labeled by α with the corresponding sum for M_j.

In addition to the equations generated in this fashion from the FSAs $M_1, \ldots, M_n$, we generate equations and inequalities reflecting the restrictions imposed by the R_j. If a

restriction language R_j is regular, we can generate equations from an FSA accepting it, exactly as described above. (Indeed, even expressions involving the shuffle and dagger operators can be handled this way: by ignoring order, shuffle can be treated as concatenation and dagger can be treated as Kleene star). Then, for each symbol $\alpha \in A_j$ and each i such that $\alpha \in \Sigma_i$, we add an equation stating that the sum of the variables for arcs labeled by α is the same in the FSA accepting R_j and in M_i, just as for symbols belonging to two FSA alphabets. In practice, we have made use of restriction languages that are simple enough that the additional inequalities can be expressed directly in terms of the variables from the M_i, avoiding the creation of many new variables and equations and reducing the size of the IP problems that must be solved. (An example of this is given below.) We have not attempted to formalize a procedure for efficiently generating inequalities from arbitrary recursive languages.

Figure 1 shows the equations for a simple concurrent program with two processes that use channels with infinite message buffers to communicate. Here, $+a$ represents the sending of a message to channel a and $-a$ represents the reception of a message from channel a. Consistent communication over a channel a is enforced by the restriction $(+a \ \ -a)^{\dagger} \otimes (+a)^{*}$ (this expression generates strings having the property that, in any prefix, the number of $-a$'s never exceeds the number of $+a$'s). From this we extract the relation that the number of $+a$ event symbols must be greater than or equal to the number of $-a$ event symbols in any string representing an execution. (This is exactly the relation that would be generated by treating the dagger as a Kleene star, but we express it in terms of the transition variables already associated with the $+a$ and $-a$ symbols.)

Every execution trace will have a corresponding solution to the inequality system we generate. However, not all solutions to the inequality system correspond to actual traces. The conditions represented by the inequality system are thus necessary, but not sufficient. There are two reasons for this. First, we ignore information about the order of event symbols given by the restriction languages. In the example of Figure 1, there is no execution in which process 1 executes $-a, +b$ and process 2 executes $-b, +a$ since any interleaving of these two strings violates the restriction for channel a or b, but since the number of events is consistent with relations derived from the restrictions, this would correspond to a solution to the inequality system. The second reason that the conditions are not sufficient is that the flow equations do not completely capture the semantics of the FSAs. The events in which a process engages in a legal execution must lie along a single path, however, the presence of cycles in the FSA graph can allow extra circular flows. An example of this is the solution to the inequalities of Figure 1 in which $x_1 = x_4 = x_6 = x_7 = f_2 = f_5 = 1$ and all other variables are zero.

The analyst usually searches for traces with certain properties by adding additional inequalities to the system. For example, an analyst might ask whether there is an execution in which more messages are sent to channel b than are received from that channel by adding the inequality $x_2 + x_3 - x_5 > 0$. Similarly, if we added transitions labeled with symbols representing permanent blocking of the process to the example, the analyst would seek executions in which a process waits forever to receive a message by adding an inequality stating that the sum of the variables corresponding to the particular symbol is greater than or equal to one.

The flow equations can also be generated directly from regular expressions. For this algorithm, picture the regular expression parsed into a tree where leaf nodes are labeled

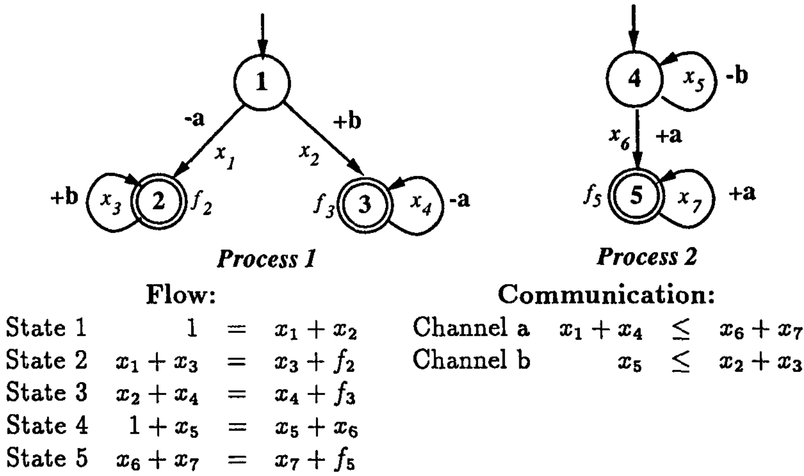

Figure 1: Example of Inequality System

with event symbols and other nodes are labeled with one of the regular operators (SEQUENCE, OR, or Kleene STAR) whose operands are its child nodes. Figure 2 shows the regular expression $(ab)^*(a \vee cb)$ so parsed. We create *environment variables* for the nodes representing the number of times the node "occurs" in the derivation of the trace (e.g., the lower left sequence node in Figure 2 occurs twice in the derivation of the string *ababcb*). Each variable has an associated *scope* consisting of a set of nodes that must occur the same number of times (e.g., a sequence node and its operands). To accomplish this, we assign an environment variable to the root node and the children of OR and STAR nodes. Then we let the scope of each environment variable be the node to which it belongs and all descendants of that node down to but not including the next node with its own environment variable. In Figure 2, the nodes with assigned variables are labeled with the variable and the nodes in the scope of a particular variable are labeled with that variable in parentheses.

We write the following equations in these variables. First, we set the variable of the root node to one, indicating that exactly one string is to be generated for this process. Second, for each OR node, we generate an equation stating that the number of times that the OR node occurs equals the sum of the numbers of times that its operand nodes occur. At STAR nodes, the appropriate inequality expressing the fact that an operand of the STAR does not occur unless the STAR node does is of the form $x_s x_o - x_o \geq 0$, where x_s is the variable associated with the STAR node and x_o is the variable associated with the operand. (Since our variables are constrained to be nonnegative, this says that x_o must be zero if x_s is.) Since solving systems of nonlinear inequalities is very much more difficult than solving linear systems, in our application of this method we have simply ignored these quadratic inequalities. (We have defined, but not used, a technique for approximating this quadratic inequality with a linear one.) Just as with cycles in the generation of equations from FSAs, this can produce solutions that will not correspond to any legitimate execution of the process. Figure 2 shows the equations generated from the given regular expression. Additional inequalities would be generated from the restriction

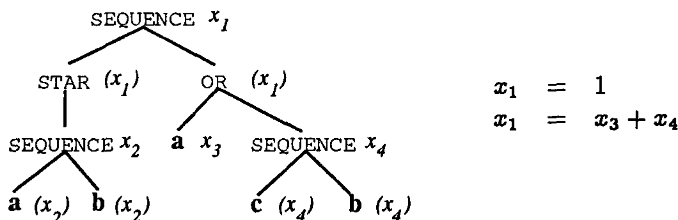

Figure 2: Example of Inequality System From Regular Expression

languages as before.

4 Implementation and Application

This method for generating inequalities reflecting necessary conditions that must be satisfied by system traces has been implemented as part of the *constrained expression toolset* [3] and applied to a variety of concurrent systems. Our experiments with the toolset clearly show that these necessary conditions are strong enough to settle many of the questions arising in analysis of concurrent systems. Furthermore, the results also show that the method can be used effectively with concurrent systems that are far too large for most other existing analysis methods to be practically applied. Here, we briefly describe the toolset, focusing on the components that generate and solve inequalities, and give some of the experimental results. More detailed descriptions of the tools and discussion of additional experiments are given in [3].

The constrained expression toolset is intended for use in analyzing concurrent system designs written in an Ada-like design notation called CEDL, and has five major components. In normal use, an analyst would first use the *deriver* to produce a constrained expression from a CEDL design. The restriction languages (called *constraints*) needed for the constrained expression representation of CEDL systems are all regular; the output of the deriver thus consists of a set of regular expressions. This constrained expression would then be supplied as input to the *constraint eliminator*, which uses regular language intersection techniques to produce an equivalent constrained expression that can be analyzed more effectively. For reasons of efficiency, the constrained expression produced by the eliminator consists of regular languages represented as regular expressions, FSAs, or in a hybrid form we call *regular expression deterministic finite automata* (REDFAs) [3, 7] that yields particularly compact systems of inequalities. The *inequality generator* then produces systems of inequalities from this constrained expression using the methods just described, augmented by inequalities representing the analyst's queries about the behavior of the concurrent system. An *integer programming package* determines whether the system of inequalities has any integer solutions and, if it does, produces one with appropriate properties (typically, in our applications, one that minimizes some measure of size). Finally, if a solution is found, a *behavior generator* performs a highly constrained reachability analysis to determine whether the solution corresponds to a trace of the concurrent system being analyzed, and to produce such a trace if one exists.

Input to the inequality generator is a constrained expression produced by the deriver from a CEDL design and possibly modified by the constraint eliminator. Each task

(asynchronous process) in the design is represented by a regular expression, an FSA, or an REDFA. The inequality generator produces a system of equations for each task, as described above. It then generates additional inequalities reflecting the constraints arising in the constrained expressions derived from CEDL designs. (For reasons of efficiency, we have chosen to sacrifice some of the language independence and build some knowledge of the CEDL constraints into the inequality generator.) The inequality generator provides a menu-driven interface that allows the analyst to formulate queries and to specify one of several objective functions for integer linear programming, and facilities that assist the analyst in interpreting the solutions found by the integer programming tool. The inequality generator is written in Common LISP. A complete description of this tool can be found in [2].

The integer programming component of the toolset is a branch-and-bound integer programming system that uses the MINOS optimization package [12] to solve the LP-relaxations of the integer programming problems. We refer to the tool that incorporates our branch-and-bound code and MINOS as IMINOS (Integer MINOS). The IMINOS tool takes an inequality system and associated objective function in the standard MPS file format as input. The input file is produced by the inequality generator. At each branch-and-bound iteration, IMINOS uses depth-first search to determine which active tree node to examine. We have experimented with three relatively naive strategies for selecting branching variables. Although we have obtained fairly good results with one of these strategies, we have begun to implement a strategy based on special ordered sets that makes use of semantic information from the constrained expression to choose branching variables [4].

We chose to base the integer programming component of the toolset on MINOS for several reasons, including the availability and robustness of MINOS and the relative ease of adding the branch-and-bound mechanism to it, despite some drawbacks. While the performance of IMINOS has been very satisfactory for demonstrating the feasibility of the general approach, further development of the toolset will require improved integer programming methods. We are therefore also investigating special-purpose algorithms for the solution of the network flow problems with side constraints that are generated by our method.

As mentioned above, the constrained expression toolset has been applied to analyze a variety of concurrent system designs, including designs for some standard problems like the dining philosophers and readers-and-writers, a protocol for distributed mutual exclusion, and an automated self-service gas station. Many of these experiments are described in [3]; here we briefly describe some of the results of experiments with several sizes and versions of the dining philosophers problem.

In the basic CEDL system, each fork and each philosopher is represented by a separate task, and a philosopher picking up or putting down a fork is modeled by a rendezvous at the up or down entry, respectively, of the fork task. Thus, in the system with n philosophers, there are $2n$ tasks. This system can deadlock when each philosopher picks up one fork. One of the standard ways to prevent this deadlock is to introduce a "host" or "butler" who ensures that all the philosophers do not attempt to eat at the same time. We have modeled this by introducing an additional host task and modifying the philosopher tasks so that a philosopher calls the `enter` entry of the host before attempting to pick up the first fork and calls the `leave` entry of the host after putting down the second fork.

	phils	tasks	size	IG time	IMINOS time	total time
basic	60	120	1141×960	158	74	629
	80	160	1521×1280	248	75	883
	100	200	1901×1600	399	120	1249
host	20	41	603×1261	157	65	467
	30	61	903×2491	538	58	1223
	40	81	1203×4121	1516	81	2941
incorrect host	20	41	607×1305	222	54	716
	30	61	905×2523	537	119	1540
	40	81	1205×4163	1603	865	4070

Figure 3: Performance on Dining Philosophers Problems

The host accepts calls at `enter` as long as no more than $n - 2$ philosophers are currently attempting to eat, and accepts calls at `leave` at any time.

The first 3 lines of the table in Figure 3 give some information about the performance of the toolset on versions of the basic system having 60, 80, and 100 philosophers when analyzed to determine whether a philosopher can be permanently blocked after picking up a fork. The columns of the table give the number of philosophers, the number of tasks, the size of the system of inequalities produced by the inequality generator (inequalities $\times$ variables), the time used by the inequality generator, the time used by IMINOS, and the total time used by the constrained expression toolset. (All times are given in CPU seconds on a DECstation 3100, and include both system and user time.) In each case, the toolset produces a system trace displaying a deadlock in which each philosopher has picked up one fork.

The next three lines of Figure 3 give the same information for 20- 30- and 40-philosopher versions of the system with host, also analyzed to determine whether a philosopher can be permanently blocked after picking up a fork. In each case, the toolset correctly reports that such blocking is impossible. In these cases, the constraint eliminator is used to modify the constrained expressions produced by the deriver in order to allow the inequality system to better reflect the dependence of control flow in the host task on the value of the variable counting the number of philosophers attempting to eat. This process, together with the additional entry calls in the philosopher tasks, results in a significantly larger system of inequalities for the dining philosophers with host than for the system without host having the same number of philosophers. For comparison, the last three lines of the figure give the results of the same analysis on systems in which the host task was modified to erroneously allow all the philosophers to attempt to eat at the same time (the condition guarding the `enter` entry was changed). In the cases with the incorrect host, the toolset produces a trace displaying deadlock.

5 Discussion

The approach to analysis of concurrent systems we have described is based on a formal model in which the possible traces of the executions of a concurrent system are represented by a language over an alphabet of event symbols. The model is powerful enough to represent all recursively enumerable sets, and has been used with a variety of design and programming languages and notations, including languages with asynchronous and

synchronous communication primitives and Petri nets. The approach involves the generation and solution of systems of inequalities that must be satisfied by all execution traces of the particular system being analyzed; the inequalities thus represent necessary, but, in general, not sufficient, conditions for a string to correspond to an execution.

The approach has been implemented as part of the constrained expression toolset, a collection of prototype tools for automated analysis of concurrent system designs written in an Ada-based design language. Experiments with this toolset, some of which are described above, have demonstrated that the necessary conditions represented by our inequalities are strong enough to answer many of the questions of interest in the analysis of concurrent systems. For example, our method of generating inequalities can be used to answer such questions as whether a component process can become permanently blocked or whether a resource will always be used in the intended mutually exclusive fashion [3]. The experiments have also shown that the method is practical for use with systems that approach, or even exceed, realistic sizes for concurrent system designs, in marked contrast with the results reported for most other analysis methods that have been implemented.

Because the inequalities represent only necessary, but not sufficient, conditions, solutions to the system of inequalities may not correspond to execution traces of the system being analyzed. Thus, when the system of inequalities is inconsistent, our method rigorously establishes that no execution of the system has the particular property of interest to the analyst. When a solution to the system of inequalities is found, however, we do not know that an execution with the desired property exists. (The behavior generator in the constrained expression toolset uses heuristic search to settle this question in many cases, as illustrated by the dining philosophers experiments reported above.) Our method is thus less accurate than methods based on construction of the full state space of the concurrent system. In general, however, the number of states that such methods must examine is exponential in the number of processes in the system. Because our method avoids constructing the full state space, it can be practically applied to much larger systems than such reachability-based methods. For example, Karam and Buhr [10] indicate that their approach "is effective for designs with a complexity in the order of 10-20 tasks" and suggest the use of a knowledge-based system for designs with 50 to 100 tasks. Similarly, Young et al. [16] suggest that a reasonable granularity for analysis of designs is "in the neighborhood of 8 processes." As indicated in the previous section, our method has been successfully used with systems having 200 or more processes.

Various methods have been proposed to reduce the complexity of determining the full state space. Among the most promising of such methods is the "stubborn sets" approach of Valmari [13]. By systematically reducing the number of states that need to be examined in order to establish a particular property of the system, this method can, for example, detect deadlock in the basic dining philosophers system in time that is linear in the number of philosophers. The range of useful application of this method is not completely clear at the present time — for the dining philosophers with host, for example, deadlock detection remains exponential in the number of philosophers.

Other methods for analyzing concurrent systems include those based on proving theorems in some logical structure associated with the system, those approaches that examine executions of a completed system or some simulation of it, and the T-invariant method proposed by Murata, Shenker, and Shatz [11]. In general, the theorem-proving methods are hard to automate, and it is difficult to assess their complexity and generality. Al-

though testing and simulation methods are relatively straightforward to implement, they are limited in the extent to which they can explore the space of potential executions of a system. This limitation, problematic even for sequential software systems, is more severe in concurrent or distributed systems, since the nondeterministic interleaving of concurrent activities in such systems dramatically increases the number of possible executions.

The method of Murata, Shenker, and Shatz [11] is very close in spirit to ours. In their approach, certain Petri nets are derived from Ada tasking programs, and the *T-invariants* of these nets are determined. These T-invariants are integer solutions to homogeneous systems of linear equations that represent necessary conditions for deadlock-free execution of the original programs, and are first used to detect and remove certain "inconsistency" deadlocks and then to guide the construction of a reachability graph to determine whether "circular" deadlocks are possible. At present, this approach has not been fully automated.

Our approach has also been extended to analyze timing properties of concurrent systems [5]. Ongoing and planned research includes the extension of our method to infinite executions, so that it can be used to answer questions about fairness and starvation, improvements in the way our method handles questions involving the order of occurrences of events, and the development of special techniques for analyzing concurrent systems containing arbitrary numbers of copies of some processes.

The approach to analysis of concurrent systems described in this paper thus has a number of advantages in comparison to other proposed approaches. It can be used with a variety of programming and design notations, and does not involve the enumeration of the full state space of the concurrent system being analyzed. Experiments with an implementation have shown that the approach can be effectively applied to systems that approach or exceed realistic sizes for concurrent system designs and that it can be used to answer a variety of interesting questions about those systems. Furthermore, by converting questions about the behavior of concurrent systems into ones of linear algebra and integer programming, our method brings a large and well-developed body of mathematical methods to bear on the concurrent system analysis problem.

Acknowledgment The work described here was conducted as part of a larger project on constrained expression analysis. We are grateful to Susan Avery, Laura Dillon, Michael Greenberg, RenHung Hwang, G. Allyn Polk, George Walden, and Jack Wileden for their contributions.

References

[1] T. Araki and N. Tokura. Flow languages equal recursively enumerable languages. *Acta Inf.*, 15:209–217, 1981.

[2] G. S. Avrunin, U. Buy, and J. Corbett. Automatic generation of inequality systems for constrained expression analysis. Technical Report 90-32, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

[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.*, November 1991, to appear.

[4] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Experiments with an improved constrained expression toolset. In *Proceedings of the Symposium on Testing, Analysis, and Verification*, Oct. 1991, to appear.

[5] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated constrained expression analysis of real-time software. Submitted for publication. Available as Technical Report 90-117, Department of Computer and Information Science, University of Massachusetts, Dec. 1990.

[6] G. S. Avrunin, L. K. Dillon, J. C. Wileden, and W. E. Riddle. Constrained expressions: Adding analysis capabilities to design methods for concurrent software systems. *IEEE Trans. Softw. Eng.*, 12(2):278–292, 1986.

[7] J. C. Corbett. On selecting a form for inequality generation in the constrained expression toolset. Constrained Expression Memorandum 90-1, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

[8] L. K. Dillon. *Analysis of Distributed Systems Using Constrained Expressions.* PhD thesis, University of Massachusetts, Amherst, 1984.

[9] L. K. Dillon, G. S. Avrunin, and J. C. Wileden. Constrained expressions: Toward broad applicability of analysis methods for distributed software systems. *ACM Trans. Prog. Lang. Syst.*, 10(3):374–402, July 1988.

[10] G. M. Karam and R. J. Buhr. Starvation and critical race analyzers for Ada. *IEEE Trans. Softw. Eng.*, 16(8):829–843, 1990.

[11] T. Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Trans. Softw. Eng.*, 15(3):314–326, 1989.

[12] M. A. Saunders. MINOS system manual. Technical Report SOL 77-31, Stanford University, Department of Operations Research, 1977.

[13] 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.

[14] J. C. Wileden. *Modelling Parallel Systems with Dynamic Structure.* PhD thesis, University of Michigan, 1978.

[15] J. C. Wileden. Constrained expressions and the analysis of designs for dynamically-structured distributed systems. In *Proceedings of the International Conference on Parallel Processing*, pages 340–344, August 1982.

[16] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification*, pages 200–209, 1989. Appeared as *Software Engineering Notes*, 14(8).