Automated Analysis of Concurrent Systems with the Constrained Expression Toolset

George S. Avrunin[•,1]
Ugo A. Buy[⋆,1]
James C. Corbett[†,1]
Laura K. Dillon[‡,2]
Jack C. Wileden[†,1,3]

COINS Technical Report 90–116
November 1990

[•]Department of Mathematics and Statistics
University of Massachusetts
Amherst, Massachusetts 01003

[⋆]Department of Electrical Engineering & Computer Science
University of Illinois
Chicago, Illinois 60680

[†]*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

[‡]Computer Science Department
University of California
Santa Barbara, California 93106

Submitted for journal publication

---

[1]Research supported in part by National Science Foundation grant CCR–8806970 and ONR grant N00014-89-J-1064
[2]Research supported in part by National Science Foundation grant CCR-8702905
[3]Research supported in part by National Science Foundation grant CCR-8704478 with cooperation from Defense Advanced Research Projects Agency (ARPA order 6104).

Automated Analysis of Concurrent Systems with the Constrained Expression Toolset

George S. Avrunin*
University of Massachusetts, Amherst

Ugo A. Buy*
University of Illinois, Chicago

James C. Corbett*
University of Massachusetts, Amherst

Laura K. Dillon†
University of California, Santa Barbara

Jack C. Wileden*,‡
University of Massachusetts, Amherst

Abstract

The *constrained expression* approach to analysis of concurrent software systems has several attractive features, including the facts that it can be used with a variety of design and programming languages and that it does not require a complete enumeration of the set of reachable states of the concurrent system. This paper reports on the construction of a toolset automating the main constrained expression analysis techniques and the results of experiments with that toolset. The toolset is capable of carrying out a completely automated analysis, starting from source code in an Ada-like design language and producing system traces displaying the properties represented by the analyst's queries. It has been successfully used with designs that approach, and in some cases exceed, realistic sizes for concurrent system designs. Limitations of the toolset and plans for future enhancements are also discussed.

*Research partially supported by NSF grant CCR-8806970 and ONR grant N00014-89-J-1064.

†Research partially supported by NSF grant CCR-8702905.

‡Research partially supported by NSF grant CCR-8704478 with cooperation from DARPA (ARPA order 6104).

1 Introduction

With increasing frequency, large software systems are organized as collections of cooperating asynchronous processes. Their size alone makes these systems hard to understand, but the difficulty is vastly increased by the introduction of nondeterminacy. Nondeterminacy in such systems can arise when the computations carried out by some components of the system depend on the unpredictable order of events occurring in other components and can also result from the deliberate use of nondeterministic program constructs. Software developers use nondeterminacy to cope with lack of knowledge about the environment, as in navigation and process control systems, to make efficient use of resources, as in operating systems, and for other reasons. Nondeterminacy is ubiquitous in both logically concurrent and truly parallel systems, but confidence in the reliability of such a system requires that its developers understand a potentially enormous number of subtle and often unexpected interactions among its components.

Developers of concurrent systems therefore need rigorous analysis methods. The analysis of concurrent software systems should begin at the design stage, so that errors can be detected early in the development process when the cost of correcting them is smallest, and continue through evaluation of completed code. Of course, different analysis methods may be appropriate at different stages of development.

A number of analysis methods for concurrent systems have been proposed, based on a variety of models of concurrent computation and intended for answering different questions at different stages of development. The methods include those based on construction of a set of possible states of the concurrent system (e.g., [1], [2], [3]), on proving theorems in some logical structure associated with the system (e.g., [4], [5], [6]), and on examining the execution of a completed system or some simulation of it (e.g., [7], [8]).

It is unlikely that any one approach will meet all the needs of developers of concurrent systems, so developers who might use these methods will need to know such things as the types and sizes of systems to which each of the methods can be usefully applied, and the sorts of questions about those systems the methods can most effectively answer. Unfortunately, we simply do not have this information for most of the proposed methods. For example, measures of the computational complexity of an analysis technique tell us something about limits on the size of the systems to which it can profitably be applied, but the complexity of many methods is not well understood. Furthermore, even a method that is known to be, say, exponential in the number of processes in the concurrent system may be able to provide useful information if the systems of interest are small enough that the method can be feasibly applied.

Further complicating the task of assessing the practical value of these methods is the fact that it is unlikely that any of them can be of much use to developers of concurrent software systems without automated support. Even high-level designs for real concurrent systems are large enough to make manual

application of rigorous analysis methods impractical, and the difficulty of the analysis usually increases as the designs are elaborated into completed code. This means that assessments of the value of analysis methods to developers of concurrent systems depend in part on the availability of implementations of those methods, and therefore on the details of those implementations.

The value of research in software engineering, however, depends on its utility as well as its elegance or intellectual fruitfulness. We therefore believe that evaluation of the potential significance of a method for analyzing concurrent software systems must include the application of an implementation of that method to a variety of types and sizes of concurrent systems, in addition to more formal and theoretical assessments. Conducting such an empirical assessment requires an implementation of the method and introduces a number of variables related more to details of the implementation and its hardware and software platforms than to the analysis method itself. But it is not possible to understand the value of the method to software developers without this sort of experience with its application.

For several years, we have been developing analysis methods based on the *constrained expression* formalism [9], [10], [11], [12], [13], [14]. The constrained expression approach to analysis has a number of attractive features. It is based on a formal model of concurrent computation that is well-suited to answering some of the natural and fundamental questions about occurrences of events that arise in analysis of concurrent systems. It can be used with a variety of standard design or programming languages based on different views of the semantics of concurrent computation and applied at different stages of the development process [12], thereby allowing developers to work in congenial and appropriate notations while retaining the ability to apply rigorous analysis methods. Furthermore, the analysis techniques limit some of the effects of combinatorial explosion, since they do not require enumeration of the set of reachable states of the system.

As we have just argued, however, an assessment of the value of the constrained expression approach for software developers requires an empirical evaluation of the methods. We have recently completed the construction of a toolset automating some of these methods and have applied it to a number of examples of concurrent systems. The purpose of this paper is to describe the toolset and the analysis methods it implements, the results of our experiments with it, and our current assessment of the strengths and weaknesses of our approach.

The paper is organized as follows. The next section gives some background on the constrained expression formalism, including a somewhat more general formulation than in our previous papers. The third section describes the tools and the analysis methods they implement. The fourth section discusses the results of our experiments with the application of the toolset. In the fifth section, we assess the strengths and weaknesses of the toolset and our approach, on both theoretical and empirical grounds. In the last two sections, we discuss our conclusions and future research plans.

2 The Constrained Expression Formalism

In the constrained expression approach to analysis of concurrent systems, the system descriptions produced during software development (e.g., designs given in some design notation) are translated into formal representations called *constrained expression representations*, to which a variety of analysis methods are then applied. This section contains a brief description of the central features of the constrained expression formalism, which was first defined in [13]. Detailed and rigorous presentations of the formalism are given in [11] and in the appendix to [12], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [10]. The description of the constrained expression formalism presented in this section generalizes aspects of these previous presentations. This more general treatment of the formalism does not affect the semantics of the constrained expressions that appear in the earlier presentations, but it may be easier to understand and it facilitates methods for composing constrained expressions and for modularizing their analysis.

Constrained expressions provide a very general model of system behaviors and have been used with a variety of descriptive notations, including a design language providing asynchronous message passing primitives, a subset of CSP (which provides synchronous message passing primitives), and Petri net languages [12]. The front-end of the constrained expression toolset described in this paper implements a particular constrained expression formulation of an Ada-like design language, called CEDL (Constrained Expression Design Language), which we use for the examples below. A reader with some familiarity with Ada should have no difficulty understanding these CEDL examples; the limitations of CEDL are discussed in Section 3.1.

2.1 Models of concurrent computation

The constrained expression formalism assumes an event-based model of computation. An execution of a concurrent system is modeled by an ordered set of event occurrences, representing the activities the system engages in and the order in which the activities occur. The complexity and duration of events depends on the level of detail at which the system is regarded. Example events might include the synchronous exchange of messages involving two processes, a process asynchronously sending (or receiving) a message to (or from) another process, a process entering its critical section, a process incrementing the value of some variable, etc. (Our usage of the "event" terminology should not be confused with that of [15], in which our events would correspond to "actions".)

Event-based models of concurrent computation can be classified according to whether they assume the event occurrences in an execution of a concurrent system are partially or totally ordered in time. In this subsection, we discuss the relationship of the constrained expression formalism to the different viewpoints associated with these assumptions. The reader who is interested in the formal-

ism only for the purpose of understanding the analysis techniques implemented by our tools may prefer to skip this discussion.

Whether the event occurrences in an execution of a system are more appropriately viewed as totally ordered or partially ordered depends on the type of system being modeled and the reason for modeling the system. When examining the execution of a complete system for which there is a notion of a global "state" or when analyzing some such simulation of a system, it is natural to hypothesize an observer who sees a set of events totally ordered by time (though an observer might see some events as simultaneous and the order of events might not be the same for all observers). In this case, what one knows about the system is obtained from these total orders, and the possible observations of a system are the fundamental objects of interest. This viewpoint is appropriate for reasoning about the possible observations of a complete system design or similar system description. It has the added advantage that the human analyst may find it easier to visualize serial executions than partially ordered ones.

Under certain circumstances, however, it may not be possible to observe a system as a whole, making it necessary to reason about an execution of the system from a log of the activities that each (sequential) process engages in. The process logs define a "precedes" relation that totally orders events occurring in an individual process. Additionally, the underlying communication primitives determine a "causes" relation between inter-process communication events. When modeling a system that uses asynchronous message passing primitives, for example, we regard the sending of a message by one process and the subsequent reception of the message by another as causally related. In general, however, the transitive closure of the precedes and causes relations defines only a partial ordering on the events occurring in a system execution. Events not related by this temporal order are (potentially) concurrent [16, 17, 15]. Partial order models of execution play an important role in the specification of concurrent systems, as they do not require that concurrent events be artificially ordered, eliminating a potential source of overspecification. A partial order model of concurrent computation, for example, provides the basis for the Unity specification language [18].

Constrained expressions have a natural interpretation in terms of a model of computation based on total ordering of event occurrences. We represent a totally ordered execution (sequence of event occurrences) by a string over an alphabet of *event symbols* defined for the given system and level of detail, with each appearance of an event symbol representing a distinct occurrence of the associated event. A string representing a totally ordered execution of a system is called a system *trace*. In this context, the constrained expression representation of a concurrent system provides a closed form representation for the set of system traces. That is, the constrained expression determines a language and this language describes the possible sequences of event symbols that can occur as system traces. This is the interpretation of constrained expressions described in our earlier work [10, 11, 12, 13].

This interpretation of constrained expressions suffices for most purposes (and, in particular, for the purpose of understanding the constrained expression analysis techniques described in this paper). However, it implies a model of computation in which event occurrences are totally ordered in any execution of the system. To explain how a constrained expression describes partial orders among system events, we show in the next subsection how the constrained expression can be used to group the system traces into *interleaving sets* [19]. Informally, an interleaving set represents a partial order on event occurrences by the full set of total orders that extend the partial order. A constrained expression can be viewed as determining a set of interleaving sets, where each interleaving set consists of those system traces that are consistent with some partially ordered execution of the system, as determined by process logs and the causes relation among communication events. (A process log corresponding to a trace is obtained by projecting the trace on the event symbols representing the events that the process participates in; the causes relation is inferred from the process logs.) The representation of an execution as an interleaving set makes it possible to express the partial order that underlies the execution: Event a occurs before event b if and only if the a-symbol precedes the b-symbol in every trace in the interleaving set. We present an example below that illustrates this interpretation.

2.2 Constrained expression representations

The constrained expression representation of a concurrent system consists of an alphabet A of event symbols and a finite collection of expressions e_i, each having an associated expression alphabet $A_i \subseteq A$, $1 \leq i \leq n$. The traditional regular expression operators (concatenation, disjunction and Kleene star), the interleave operator (which is regular), and the transitive closure of the unary interleave operator (which is not regular and is also called the *dagger* operator), are used for forming the component expressions. Intuitively, A defines the alphabet used for describing system traces (as strings), and each component expression e_i specifies the patterns of symbols from A_i that appear in system traces. More precisely, we say a string *satisfies* an expression if projecting the string on the expression alphabet produces a string in the language of the expression, and *violates* the expression otherwise. We then consider any string that satisfies all the component expressions in the constrained expression representation of a system to represent a system trace. In other words, those strings s over the alphabet A for which the projection of s on A_i lies in the language of e_i, for $i = 1 \ldots n$, represent system traces.

In the case of a design written in CEDL, our Ada-like design language, each component sequential process T, called a *task* in the Ada terminology, gives rise to a *task expression* e_T and corresponding task alphabet A_T. The task expression describes the activities in which the task engages. However, because a task expression is derived from the code for a single task, it does not reflect the ac-

```
task F0 is
     entry U0;  -- pick up fork 0
     entry D0;  -- put down fork 0
end F0;

task body F0 is
begin
    loop
      accept U0;  -- pick up
      accept D0;  -- put down
    end loop;
end F0;
```

```
task P0;

task body P0 is
begin
    while ... loop
      ... ;      -- Think
      F1.U1;
      F0.U0;
      ... ;      -- Eat
      F1.D1;
      F0.D0;
    end loop;
end P0;
```

Figure 1: Fork and Philosopher Tasks from Dining Philosophers in CEDL

tivities of other tasks, and so it does not express restrictions imposed on a task's activities by the environment in which it executes. Moreover, certain aspects of the semantics of a design or programming notation, such as the patterns in which variables can be used and modified, are more easily expressed in separate expressions. For these reasons, the constrained expression representation for a system usually contains some additional expressions e_j and corresponding expression alphabets A_j. We call these expressions *constraints*, since they further restrict the patterns of symbols appearing in system traces. Constraints are typically derived from the full system description and may relate symbols from different task alphabets.

We consider a CEDL version of the dining philosophers problem with three philosophers to illustrate these ideas. (A fuller description of this problem is given in Section 4.) Figure 1 shows the CEDL code for one fork task and one philosopher task from this system. The fork task loops repeatedly, accepting calls to its `U0` and `D0` entries. The philosopher task loops an indeterminate number of times (as indicated by the elided test in the `while` statement), calling the `U` entries of the fork tasks on its "left" and "right" and then calling the `D` entries of those tasks. There are two more fork tasks and two more philosopher tasks in the system, with similar designs. Figure 2 gives the task expressions produced by the toolset for these two tasks. We show expressions in the LISP-like prefix notation used as input to several of the tools, which uses "`NIL`" to denote the empty string, "`SEQUENCE`" for the concatenation operator, "`OR`" for disjunction, and "`STAR`" for the Kleene star. For the example, we assume the set of symbols appearing in an expression determines its alphabet. The table in Figure 3 summarizes the interpretation of event symbols. We note that the permanent blocking of a task indicated by the hang symbols does not presuppose any particular cause for this blocking, which could be due to circular deadlock, termination of other tasks, or other reasons. Figure 4 shows a constraint that

```
(deftask f0
   ("SEQUENCE" "beg_loop(f00)"
      ("STAR"
         ("SEQUENCE"
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.u0)" "end_rend(p1;f0.u0)" )
               ("SEQUENCE" "beg_rend(p0;f0.u0)" "end_rend(p0;f0.u0)" ))
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.d0)" "end_rend(p1;f0.d0)" )
               ("SEQUENCE" "beg_rend(p0;f0.d0)" "end_rend(p0;f0.d0)" ))))
      ("OR"
         ("SEQUENCE" "hang_a(f0.u0)" "stop(f0)" )
         ("SEQUENCE"
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.u0)" "end_rend(p1;f0.u0)" )
               ("SEQUENCE" "beg_rend(p0;f0.u0)" "end_rend(p0;f0.u0)" ))
            "hang_a(f0.d0)" "stop(f0)" ))))


(deftask p0
   ("SEQUENCE" "beg_loop(p00)"
      ("STAR"
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "call(p0;f1.d1)" "resume(p0;f1.d1)"
            "call(p0;f0.d0)" "resume(p0;f0.d0)" ))
      ("OR"
         ("SEQUENCE" "end_loop(p00)" "term(p0)" )
         ("SEQUENCE" "hang_c(p0;f1.u1)" "stop(p0)" )
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "hang_c(p0;f0.u0)"
            "stop(p0)" )
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "hang_c(p0;f1.d1)" "stop(p0)" )
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "call(p0;f1.d1)" "resume(p0;f1.d1)"
            "hang_c(p0;f0.d0)" "stop(p0)" ))))
```

Figure 2: Two Task Expressions Derived From the Dining Philosophers Problem

Symbol	*Associated Event*
beg_loop(L)	Begin execution of loop L
beg_rend(T,E)	Begin rendezvous with task T on entry E
call(T,E)	Task T calls entry E
end_loop(L)	End execution of loop L
end_rend(T,E)	End rendezvous with task T on entry E
hang_a(E)	Task is permanently blocked waiting to accept a call on entry E
hang_c(T,E)	Task T is permanently blocked calling entry E
resume(T,E)	Resume T, after rendezvous on entry E
stop(T)	Task T stops execution (abnormal termination)
term(T)	Task T terminates (normally)

Figure 3: Interpretation of Event Symbols

```
(defconstraint SYNCHRONIZATION_1
   ("SEQUENCE"
      ("STAR"
         ("SEQUENCE" "call(p0;f1.u1)" "beg_rend(p0;f1.u1)" "end_rend(p0;f1.u1)"
            "resume(p0;f1.u1)" ))
      ("OR"
         "NIL"
         ("SEQUENCE" "call(p0;f1.u1)" "beg_rend(p0;f1.u1)" ))))
```

Figure 4: A Constraint Used to Enforce Proper Synchronization of Rendezvous at a Particular Entry

```
(defconstraint BLOCKING_1
   ("OR"
      "NIL"
      "hang_a(f0.u0)"
      ("STAR"
         ("OR" "hang_c(p1;f0.u0)" "hang_c(p0;f0.u0)" ))))
```

Figure 5: A Constraint Used to Ensure That Two Tasks Do Not Wait Forever to Rendezvous with Each Other at a Particular Entry

```
(defconstraint QUEUEING_1
   ("STAR"
     ("OR"
        ("SEQUENCE" "call(p1;f0.u0)"
           ("STAR" "call(p0;f0.u0)" "beg_rend(p1;f0.u0)" "call(p1;f0.u0)"
              "beg_rend(p0;f0.u0)")
           "beg_rend(p1;f0.u0)")
        ("SEQUENCE" "call(p1;f0.u0)"
           ("STAR" "call(p0;f0.u0)" "beg_rend(p1;f0.u0)" "call(p1;f0.u0)"
              "beg_rend(p0;f0.u0)")
           "call(p0;f0.u0)" "beg_rend(p1;f0.u0)" "beg_rend(p0;f0.u0)")
        ("SEQUENCE" "call(p0;f0.u0)"
           ("STAR" "call(p1;f0.u0)" "beg_rend(p0;f0.u0)" "call(p0;f0.u0)"
              "beg_rend(p1;f0.u0)")
           "beg_rend(p0;f0.u0)")
        ("SEQUENCE" "call(p0;f0.u0)"
           ("STAR" "call(p1;f0.u0)" "beg_rend(p0;f0.u0)" "call(p0;f0.u0)"
              "beg_rend(p1;f0.u0)")
           "call(p1;f0.u0)" "beg_rend(p0;f0.u0)" "beg_rend(p1;f0.u0)"))))
```

Figure 6: A Constraint Used to Ensure That Entry Queues Are Serviced on a First-Come-First-Served Basis

enforces proper synchronization of rendezvous for one of the entries of a fork task. Similar sychronization constraints are required for all entries. Figure 5 shows a constraint that ensures a task does not wait forever for a rendezvous with another task if the second task is also waiting for the same rendezvous. Figure 6 shows a constraint that ensures the order in which two philosopher tasks call the same entry of a fork task determines the order in which the fork task accepts the calls. Other types of constraints that do not occur in this example enforce the correct dependence of control flow on the values of variables and handle the failure of nested rendezvous. An example of the former type is presented in section 3 (see Figure 9).

Two system traces produced from the constrained expression representation of a CEDL design can be regarded as describing the same partially ordered execution if they have identical projections on each of the task alphabets (i.e., identical task logs). In the set of interleaving sets model, this means that the traces belong to the same interleaving set. Consider, for example, an execution of the dining philosophers system in which two adjacent philosophers, say, P0 and P1, each think and eat once (execute the bodies of their loops once) and the remaining philosopher never does anything (the elided test at the beginning of its loop fails immediately). In any such execution, P0 attempts to pick up fork F1 and then fork F0, while P1 first attempts to pick up F2 and then F1. The system admits two partially ordered executions in which P0 and P1 each eats once and P2 does nothing, corresponding to the two possible orders in which

P0 and P1 can pick up their common fork F1. This is reflected in the fact that traces describing such executions may produce one of two possible projections on the alphabet of F1. If P1 picks up F1 first, then P0 must wait for P1 to put F1 down before picking it up, and so the interleaving set that corresponds to this execution contains a single system trace. However, if P0 picks up F1 first, then the CEDL code does not serialize the philosophers' use of their other forks, and there are traces in the interleaving set that describe different orderings in the use of these forks.

The descriptions of the constrained expression formalism in our previous papers provide a more operational, but also less general, characterization of the set of system traces defined by the constrained expression representation of a distributed system. These descriptions outline a two-step procedure for generating system traces, in which *candidate* traces are first generated by interleaving strings from the languages of the task expressions, and then those candidates that do not satisfy the constraints are filtered out, leaving only traces that describe legitimate system behaviors. This characterization of a system trace is consistent with the characterization above, provided that the alphabets of the task expressions are disjoint; as long as the task alphabets are disjoint, a candidate trace will automatically satisfy the task expressions, and need only be checked against the constraints. The more general characterization of constrained expressions described in this paper treats task expressions and constraints more uniformly, making it easier to compose constrained expressions in a manner that is appropriate for modularizing the representation and analysis of systems. Section 6 outlines ongoing research that is producing more modular analysis methods.

2.3 Constrained expression analysis

Our main constrained expression analysis techniques require that questions about the behavior of a concurrent system be formulated in terms of whether a particular event symbol, or pattern of event symbols, occurs in a system trace. In the dining philosophers, for example, the question of whether a philosopher that has finished thinking can be blocked indefinitely from eating can be phrased in terms of the occurrence of hang_c symbols representing the permanent blocking of the philosopher task on a call to one of the appropriate entries of the fork tasks. The relevant questions to ask about a system, of course, depend on the particular system being analyzed and the correctness criteria for that system.

Starting from the assumption that the specified symbol, or pattern of symbols, does occur in a system trace, we use the form of the task expressions and constraints to generate inequalities involving the number of occurrences of various event symbols in segments of the trace. If the system of inequalities thus generated is inconsistent, the original assumption is incorrect and the specified symbol or pattern of symbols does not occur in a legal system trace. If the inequalities are consistent, we use them in attempting to construct a system

trace containing the specified pattern. The next section describes this very general approach to analysis in more detail and explains how we have automated important aspects of the analysis.

3 The Tools

There are five major components of the constrained expression toolset (see Figure 7). In normal use, an analyst would first use the *deriver* to produce a constrained expression representation from a concurrent system design written in the CEDL design language. This constrained expression would then be used as input to the *constraint eliminator*, which intersects some of the task expressions and constraints, producing an equivalent constrained expression with fewer constraints. The reasons for this procedure are explained below. The *inequality generator* takes the constrained expression produced by the eliminator as its input, together with a query formulated by the analyst, and produces a system of linear inequalities capturing certain features of the constrained expression and the query. These inequalities involve variables representing the structure of the task expressions and the numbers of occurrences of particular events in the traces or behaviors of the concurrent system being analyzed. The IMINOS *integer programming package* would then be used to determine whether this system has any integer solutions and, if it does, to find one with appropriate properties. The inequality generator provides facilities to assist the analyst in interpreting the system of inequalities and the solution, if any, found by IMINOS. When a solution is found, the *behavior generator* uses heuristic search techniques to determine whether this solution corresponds to an actual system trace, and to produce such a trace if it does. The behavior generator can also be used with information about a candidate trace provided by the analyst.

In the remainder of this section, we discuss each of the components of the toolset in more detail.

3.1 The deriver

The deriver [20] provides a front-end for the constrained expression toolset. It translates system designs into constrained expressions, which are then manipulated and analyzed by various other tools. This allows software developers to use more congenial design notations when writing designs and, at the same time, permits analysts to subject those designs to rigorous analysis for behavioral properties.

Our current deriver requires that designs be written in an Ada-like design language, called CEDL (Constrained Expression Design Language). CEDL focuses on the expression of communication and synchronization in a concurrent system, and language features not related to concurrency are kept to a minimum. The most important limitations of CEDL designs can be summarized as

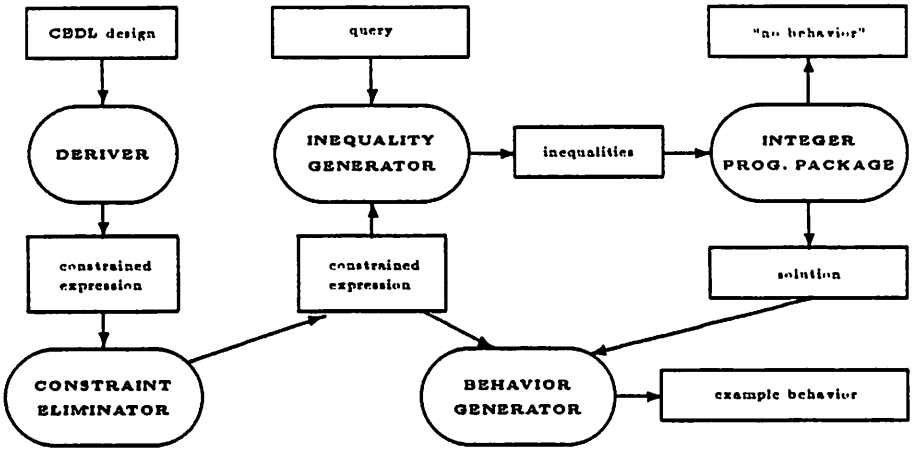

Figure 7: Diagram of Constrained Expression Toolset

follows:

- `Boolean` is the only predefined type; all other types are specified using enumeration types.
- There are no global variables.
- There are no primitives for data encapsulation. Packages simply group together type and variable declarations, all of which are exported.
- Design units may not be generic.
- There are no exception handling features.
- Design units may not be nested.
- There are no input (`get`) or output (`put`) statements.

The restriction against nesting, besides simplifying the constrained expression representations for CEDL designs, reflects our belief that nesting is a poor design (and programming) practice [21]. Other restrictions limit the complexity of CEDL designs and of their constrained expression representations. Most of the Ada control-flow constructs have correspondents in CEDL. CEDL also

provides an ellipsis notation (written "...") for expressing incompleteness in designs. The use of this construct was illustrated in the dining philosophers example of Section 2.2. The incompleteness construct can be used to elide statements, expressions, declarations and types that will be elaborated in later system descriptions.

The deriver produces task expressions for each of the tasks in a CEDL design from the code for the task bodies. Our original constrained expression formulations of different development notations associated *translation rules* with statement types. In the translation rule approach, the task expression for a task is obtained by concatenating together expressions produced by applying the appropriate translation rules to the statements that appear in the task body. The conceptual simplicity of this approach is pleasing. However, in practice, the translation rule approach requires overly complex translation rules involving "pseudo-event symbols" (event symbols that do not represent actual system events) and special constraints describing the sequencing of pseudo-event symbols in order to represent executions in which tasks are permanently blocked or stop executing prematurely for other reasons (such as the commission of run-time errors). For this reason, we now use *attribute grammars*, rather than translation rules, for generating task expressions. We associate attributes with CEDL statements that, collectively, give the translations for the statements, and that can be appropriately combined, when the statements appear within a compound statement, to produce the attributes for the compound statement. Attribute equations define the synthesis of attribute values in the standard manner [22]. The attribute grammar approach allows us to produce constrained expression representations for CEDL designs that are much simpler than the representations described in our earlier papers [11, 23, 12]. Figure 2 shows two task expressions produced by the deriver from the CEDL code of Figure 1.

The deriver produces the constraints for the constrained expression representation of a CEDL design by instantiating a fixed set of constraint templates. The instantiation of the constraint templates requires information such as the names of the tasks that call an entry, the type of a variable and the values associated with that type, which is accumulated as the design is parsed. Figures 4, 5, and 6 give examples of the constraints produced by the deriver.

The current deriver is written in Ada, and was developed using versions of standard compiler construction tools produced by the Arcadia project on software development environments [24] and Arcadia's Graph Definition Language and GRAPHITE processor [25]. The deriver generates a graph representing the constrained expression and translates this into the LISP-like notation used by most of the other tools.

The deriver is, of course, specific to CEDL. In principle, the other tools could be constructed in a CEDL-independent fashion, and used with constrained expressions produced from any design notation. In practice, as discussed below, several of the other tools rely on certain assumptions that are dependent on the features of CEDL in order to improve efficiency.

3.2 The constraint eliminator

As discussed in the next subsection, the inequalities we generate do not express the full semantics of constrained expressions, with the result that there may be solutions to the inequalities that do not correspond to system traces. In particular, the inequalities do not express certain restrictions on system traces that involve only the order in which certain events occur, rather than the numbers of such events in the traces. In practice, the most significant of these restrictions are those imposed by the constraints that ensure the consistent use of variables in CEDL programs. Without taking such restrictions into account, we would get solutions to our inequalities corresponding to "traces" in which, for example, the `else` branch of an `if` statement is taken even though the Boolean condition of the `if` statement evaluates to `true`. We use the constraint eliminator [26] to modify the constrained expression representations in such a way that the inequalities generated from them exclude such solutions. The tool can also be used for more general modifications of the constrained expressions, as discussed below.

To see how the constraint eliminator is used, consider the following segment of a task `T`:

```
flag := ...;
if flag then
  S.A;
end if;
if not flag then
  S.B;
end if;
```

Figure 8 shows the portion of the task expression for task `T` corresponding to this fragment. This segment should always call exactly one of entries `A` or `B` of task `S`; however, the task expression produced by the deriver permits system traces in which both calls are made and traces in which neither call is made. In the full constrained expression representation, the *dataflow constraint* shown in Figure 9 filters out the strings in which both of these calls occur and the strings in which neither call occurs. The constraint allows any number of `def(flag;`*val*`)` symbols, each of which represents the assignment of the value *val* to the variable `flag` and can be followed by any number of `use(flag;`*val*`)` symbols with that particular value, each representing a use of the variable. Any string satisfying both the task expression, with its `use(flag;_)` symbols in the various alternatives, and the constraint will involve exactly one of the entry calls.

The constraint eliminator modifies the constrained expression so that each of the resulting task expressions already incorporates any constraints involving only symbols from that task (i.e., any string satisfying the new task expression

```
("SEQUENCE" ("OR" "def(flag;true)"
                 "def(flag;false)")
            ("OR" ("SEQUENCE" "use(flag;true)" "call(T;S.A)" "resume(T;S.A)")
                 "use(flag;false)")
            ("OR" ("SEQUENCE" "use(flag;false)" "call(T;S.B)" "resume(T;S.B)")
                 "use(flag;true)"))
```

Figure 8: Part of the Task Expression for Task T

```
("STAR" ("OR" ("SEQUENCE" "def(flag;true)" ("STAR" "use(flag;true)"))
              ("SEQUENCE" "def(flag;false)" ("STAR" "use(flag;false)"))))
```

Figure 9: Dataflow Constraint for Local Variable `flag`

satisfies both the old task expression and the constraints). For example, Figure 10 shows the result of incorporating the dataflow constraint for the variable `flag` into the task expression for task T shown above. The inequalities generated from the resulting task expression then reflect the restrictions imposed by the constraint, and do not admit solutions corresponding to violations of that constraint.

The constraint eliminator takes a set of task expressions and constraints as input. Each constraint whose alphabet involves only symbols from a single task alphabet (an *intra-task* constraint) is incorporated into the task expression it constrains and is then removed. The resulting set of task expressions and constraints is output. The task expressions incorporating their intra-task constraints may be output either as regular expressions (REs), deterministic finite automata (DFAs), or in a hybrid form we call regular expression deterministic finite automata (REDFAs). REDFAs are DFAs whose arcs are labeled with regular expressions satisfying the following conditions:

1. Given a regular expression e, define $FIRST(e)$ to be the set of symbols that can begin strings generated by e. For each state in the REDFA with transitions on the expressions $e_1, e_2, \ldots, e_n$,

$$FIRST(e_i) \cap FIRST(e_j) = \emptyset,\ i \neq j$$

```
("OR" ("SEQUENCE" "def(flag;true)" "use(flag;true)" "call(T;S.A)"
                  "resume(T;S.A)" "use(flag;true)")
      ("SEQUENCE" "def(flag;false)" "use(flag;false)" "use(flag;false)"
                  "call(T;S.B)" "resume(T;S.B)"))
```

Figure 10: Part of Task Expression After Elimination of Dataflow Constraint

2. Each expression labeling an arc in the REDFA is deterministic in the following way: For each disjunction of subexpressions $e_1 \vee e_2 \vee \ldots e_n$ in the expression,

$$FIRST(e_i) \cap FIRST(e_j) = \emptyset,\ i \neq j$$

We have found that it is easier to generate "efficient" inequality systems from REs, but that after constraint elimination, the REs for some tasks are very much larger than their corresponding DFAs. Here, the efficiency of an inequality system is, roughly speaking, the size of the task representation (RE, DFA, or REDFA), in bytes, say, divided by the size of the inequality system (variables × inequalities). Unlike REs, REDFAs are never significantly larger than the DFAs they are generated from, and unlike DFAs, very efficient inequality systems can be generated from them easily. A more complete discussion of the relative advantages of inequality generation from REs, DFAs, and REDFAs appears in [27].

To incorporate a set of intra-task constraints into a task expression, all the regular expressions involved are converted to DFAs, which are then intersected pairwise. The intersection of two DFAs with state sets S_1 and S_2 is performed by generating, in depth-first order, the subset of $S_1 \times S_2$ reachable from the state pair containing the start states of the DFAs. The intersection differs from standard DFA intersection in the following way: At each state of a DFA, we assume implicit self-loops on all symbols not appearing in the alphabet of that DFA. This allows the DFA representing a constraint to accept symbols not in its alphabet without changing state. Assuming the constraint alphabet is a subset of the task alphabet, the result of the intersection is a DFA that accepts exactly those strings accepted by the original task DFA in which the symbols contained in the intra-task constraints appear in the order required by those constraints. In the case of a dataflow constraint for a local variable, this essentially encodes the value of the variable into the DFA state (where before the state encoded only the syntactic location within the task design), usually increasing the number of states in the task DFA, but guaranteeing consistent use of the variable. In CEDL, the intra-task constraints are exactly the dataflow constraints since there are no global variables and all other constraints involve more than one task expression.

Using the special intersection defined above, the constraint eliminator could, theoretically, intersect all the tasks and constraints, producing one large DFA whose language is the set of legal traces of the concurrent system. While this would prevent violation of all the constraints (not just the intra-task ones), the resulting DFA would be similar to a reachability graph of the concurrent system, and equally large — in the worst case exponential in the number of tasks [28]. It is exactly this state explosion we seek to avoid by considering the tasks separately and ignoring some of the dependency (that relating to order) between them.

The constraint eliminator is written in Common LISP. It defines an abstract DFA data type and a set of general purpose functions for manipulating REs and DFAs (e.g., converting REs to DFAs, intersecting DFAs) that are also used by the inequality generator and behavior generator tools. The general intersection function was designed to support experiments with methods being developed for modularizing constrained expression analysis, as described in Section 6.

3.3 The inequality generator

The analysis implemented by the constrained expression toolset involves the generation of a system of linear inequalities expressing features of both the constrained expression representation of the concurrent system being analyzed and a query posed by the analyst. We now describe the inequality generator component of the toolset. A complete description of this tool is given in [29].

The input to the inequality generator consists of a list of tasks. The tasks may be represented as regular expressions, or, following constraint elimination, as DFAs or REDFAs. For each task, the inequality generator produces a collection of equations. It then generates additional inequalities reflecting part of the semantics of certain of the constraints. The generation of equations for the tasks depends only on the basic structure of regular expressions and finite state automata, but the generation of inequalities from constraints depends on features of CEDL. In principle, since the CEDL constraints are all regular expressions, the generation of inequalities from tasks and constraints could be accomplished in a uniform manner. While this would be more consistent with the interpretation of the semantics of constrained expressions given in Section 2, the separate procedure we have adopted in the inequality generator improves the efficiency of the tool and reduces the size of the systems of inequalities it produces, as discussed below.

We begin by describing the generation of equations from the tasks, first from regular expressions and then from DFAs and REDFAs, and then discuss generation of inequalities reflecting constraints.

The basic idea behind the generation of inequalities reflecting the constrained expression is as follows. The semantics of regular expressions implies that each operand of a `SEQUENCE` operator must occur the same number of times, that the sum of the number of occurrences of the operands of an `OR` operator must equal the number of "occurrences" of the operator itself, and that, if the operand of a Kleene star operator occurs at all, the number of its occurrences is unrestricted. Of course, this interpretation does not fully capture the information contained in the regular expression about the order in which the operands occur. Given a regular expression, we build a parse tree in which each nonterminal node is an operator and each terminal node is an event symbol. Assigning a variable in the integer programming problem to each node to represent the number of times we pass through that node in generating a string from the regular expression, the observations above give a linear equation at each `SEQUENCE` or `OR` node, and

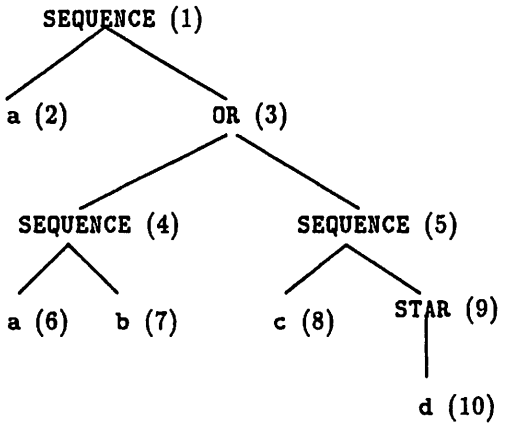

Figure 11: Parse Tree for the Regular expression $a(ab \vee cd^*)$.

a quadratic inequality at each STAR node. (The quadratic inequality is of the form $x_s \cdot x_o - x_o \geq 0$, where x_s is the variable associated to the STAR node and x_o is the variable associated to the operand of the STAR; since all our variables are constrained to be nonnegative, this inequality says that x_o must be zero if x_s is.) We also generate an equation setting the value of the variable associated with the root node of the parse tree to one, representing the fact that the task begins execution exactly once.

This approach is illustrated with the example in Figure 11, which gives a parse tree for the regular expression $a(ab \vee cd^*)$. (The letters a, b, c, and d stand for event symbols in a task expression.) The number in parentheses at each node gives the index of the variable corresponding to that node. The following inequalities would be generated from this parse tree.

$$\begin{aligned} x_1 &= 1 \\ x_1 - x_2 &= 0 \\ x_1 - x_3 &= 0 \\ x_3 - x_4 - x_5 &= 0 \\ x_4 - x_6 &= 0 \\ x_4 - x_7 &= 0 \\ x_5 - x_8 &= 0 \\ x_5 - x_9 &= 0 \\ x_9 x_{10} - x_{10} &\geq 0 \end{aligned}$$

In general, of course, quadratic integer programming problems are much harder to solve than linear ones, and we have therefore chosen simply to ignore the inequalities that should be generated at STAR nodes. (In fact, if the variables are all assumed to be bounded above by, say, B, we can achieve the effect of the quadratic inequalities with linear ones of the form $x_o \leq B \cdot x_s$. We do not use this technique routinely. We have used it in certain special cases, as described in Section 4.)

In this fashion, we generate a system of linear inequalities from the task expressions. Our first prototype of the inquality generator [30] used exactly this approach. The current inequality generator makes use of several optimizations that significantly reduce the number of inequalities and variables required. For example, all the operands of a SEQUENCE operator occur the same number of times, so it is not necessary to generate separate variables for each of them, together with equations stating that these variables take values equal to that of the SEQUENCE node. Our experience is that such optimizations reduce the numbers of both inequalities and variables generated from a regular expression by a factor of about six.

To generate inequalities from a DFA or REDFA representation of a task expression, we can assign a variable to each arc, rather than each node, and an extra variable to each accepting state. We then generate a "flow" equation for each state requiring that the sum of the variables corresponding to arcs into that state must equal the sum of the variables corresponding to arcs leaving the state, except that at the initial state we require the sum of the variables on incoming arcs to be equal to the sum of the variables on outgoing arcs minus one, and we count the extra variables for the accepting states as if they corresponded to outgoing arcs. For REDFAs, some arcs are labelled by regular expressions rather than single event symbols. For each such arc, we generate additional equations corresponding to the regular expression labelling that arc, using the method described above, but associating the variable corresponding to the arc with the root node of the parse tree of the regular expression. Figure 12 shows a DFA accepting the language of the regular expression of Figure 11. The numbers in parentheses next to the arcs and accepting states give the indices of the corresponding variables. The equations generated from this DFA are:

$$\begin{aligned} x_1 &= 1 \\ x_1 - x_2 - x_3 &= 0 \\ x_2 - x_4 &= 0 \\ x_4 - x_6 &= 0 \\ x_3 + x_5 - x_5 - x_7 &= 0 \end{aligned}$$

Having produced equations for each task, the inequality generator then be-

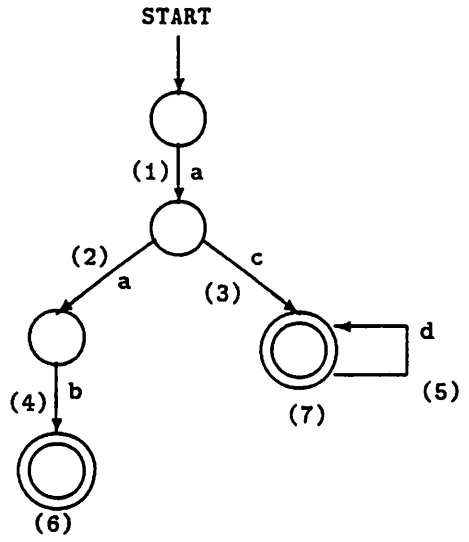

Figure 12: DFA Accepting the Language of $a(ab \vee cd^*)$

gins to generate linear inequalities reflecting some of the constraints. The constraints impose restrictions on the order and number of occurrences of event symbols in traces of the system. The integer programming variables we use only involve the total number of occurrences of symbols (or, more precisely, of traversals of nodes in the parse trees or arcs in the finite state automata), and do not reflect the order in which those symbols occur. We therefore wish to extract the information about total numbers of occurrences of event symbols from the constraints.

Note first that the total number of occurrences of a particular event symbol is given by the sum of certain variables in the equations generated from the tasks. If the symbol occurs in a task represented by a regular expression, the number of occurrences of the symbol is equal to the sum of the variables corresponding to the terminal nodes at which that symbol appears. Thus, in the example of Figure 11, the number of occurrences of the event symbol represented by a is $x_2 + x_6$. If the symbol occurs in a task represented by a DFA, the number of occurrences is given by the sum of the variables corresponding to those arcs labeled by the symbol, while in the case of a task represented by an REDFA, the number of occurrences may involve variables associated with both arcs and nodes in the parse trees of the regular expressions labeling those arcs. The inequality generator maintains a table giving the number of occurrences of each event symbol as a sum of integer programming variables.

To see how information involving the number of occurrences of event symbols is obtained from constraints, consider first the synchronization constraint shown in Figure 4. In any string satisfying this constraint, the number of `call(p0;f1.u1)` symbols must equal the number of `beg_rend(p0;f1.u1)` symbols, and the number of `end_rend(p0;f1.u1)` symbols must equal the number of `resume(p0;f1.u1)` symbols. We may therefore generate equations involving the sums of variables corresponding to the numbers of occurrences of these symbols. The constraint further requires that the various symbols occur in a specified order, but this fact cannot be expressed in terms of the integer programming variables associated with the tasks. Similarly, from the constraint of Figure 5 and the fact that task expressions produced by the deriver have the property that each task contributes exactly one hang symbol to a trace, we conclude that the sum of the number of `hang_a(f0.u0)` symbols and the number of `hang_c(pi;f0.u0)` symbols cannot exceed one, for $i = 0, 1$. Other inequalities are obtained from the constraints that deal with the failure of nested rendezvous. (The constraints that enforce the queueing of entry calls and the dependence of control flow on data involve only the order in which event symbols occur and not the total number of their occurrences, and are ignored in this part of the analysis. The constraint eliminator takes those constraints involving intratask dataflow into account before inequalities are generated.) As noted above, it would be possible to generate inequalities from a constraint by first generating equations from the regular expression, as we do for task expressions, and then generating equations stating that the number of occurrences of an event symbol coming from the task in which it appears must equal the number coming from each constraint in which it appears. This approach, though pleasingly uniform and language-independent, would lead to the introduction of many additional variables and equations coming from the constraints. We have therefore chosen to sacrifice some of the language-independence and build some knowledge of the CEDL constraint templates into the inequality generator.

We thus generate a system of inequalities reflecting a large part, but not all, of the semantics of the constrained expression representation. Queries about the behaviors of the concurrent system are also expressed in terms of the integer programming variables. For example, an analyst could formulate the statement that a philosopher is permanently prevented from eating as an equality stating that at least one of certain `hang_c` symbols occurs (i.e., that the sum of certain variables is one). Adding this to the system of inequalities obtained from the constrained expression, we would obtain a system reflecting both the constrained expression and the query. If this system has no integral solution, then the CEDL system has no trace in which a philosopher task waits indefinitely for a rendezvous with a fork task. If there is an integral solution, this does not guarantee that a behavior of the CEDL system exists in which the philosopher task waits indefinitely — we have ignored information about order in generating our inequalities, so the solution may be "spurious" in the sense that it does not correspond to an actual behavior. But we can use the event counts obtained

from the solution as a guide in searching for a real behavior with the property expressed in the query.

The inequality generator provides a menu-driven interface allowing the analyst to formulate queries using event symbols, rather than only integer programming variables, and it allows the analyst to specify one of several objective functions for integer linear programming. It also provides facilities that assist the analyst in interpreting the systems of inequalities and solutions found by the integer programming tool in terms of the task expressions and constraints. It is written in Common LISP, and uses some of the abstract data types and functions provided by the constraint eliminator.

3.4 IMINOS

We solve the inequality systems produced by our inequality generator using a branch-and-bound algorithm employing the variable dichotomy scheme first introduced by Dakin [31]. Our implementation of this algorithm makes use of the MINOS [32] optimization package to solve LP-relaxations of the integer programming problems. We refer to the tool that incorporates our code and MINOS as IMINOS (Integer MINOS) [33]. The IMINOS tool takes an inequality system and associated objective function in the standard MPS file format as input. This input file is produced by the inequality generator.

The central idea behind the branch-and-bound algorithm is to enumerate the nodes of a tree of linear programming (non-integer) problems. If the optimal solution to one of these problems is integral, it is a solution to the original integer programming problem (though possibly not an optimal one), and the node represented by that linear programming problem is a terminal one. If the optimal solution to one of these linear programming problems is not integral, the algorithm creates two successor nodes by adding particular inequalities to the system. For instance, if variable x_i has value 3.5 in the optimal solution of one of the linear programming problems, we create two new systems by adding one of the inequalities $x_i \leq 3$ or $x_i \geq 4$. The optimal values of the objective function cannot increase as the tree is descended, so the objective values at integer solutions found earlier can be used to prune the tree. Additional details on this algorithm may be found, for example, in [34].

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, and have obtained the best results with a round-robin method in which the search for a nonintegral value starts with the variable last used as a branching variable. With the other strategies, we have encountered problems with cycling, in which the program branches alternately on two variables and never finds a solution in which all variables take integral values. Cycling is possible with the third strategy, but we have encountered it only infrequently. As discussed in Section 6, we are planning to replace this strategy with one that makes use of semantic information

from the constrained expression to choose branching variables.

We chose to base the integer programming component of our toolset on MINOS for several reasons, including the availability and robustness of the MINOS system and the relative ease of adding the branch-and-bound mechanism to it. Disadvantages, for our purposes, are that MINOS implements only a primal algorithm, requiring simplex iterations to reattain feasibility when additional inequalities are added in the branch-and-bound process, and that it is a general-purpose package that does not take advantage of the special structure of our systems. Although the performance of IMINOS has generally been very satisfactory despite these disadvantages, as indicated by the results discussed in Section 4, some problems have arisen with large systems of inequalities. As briefly discussed in Section 6, we are therefore investigating approaches to integer programming that take advantage of the fact that our systems of inequalities can be regarded as network problems with side constraints.

3.5 The behavior generator

If IMINOS has produced a solution to the system of inequalities, the next step is to determine whether that solution corresponds to a trace of the concurrent system being analyzed. This is the principal function of the behavior generator [35]. Given the solution and the constrained expression (a set of task REs, DFAs, or REDFAs along with constraints) as input, the behavior generator will attempt to construct a system trace using the information in the solution as a guide. This information consists of total event counts for every event symbol and also includes counts for each arc in the DFA representation of the task — provided that the inequalities for that task were generated from either the DFA or REDFA form of the task.

The behavior generator performs a highly constrained reachability search on the global state space of the concurrent system. The global state space is, in general, exponential in the number of tasks, but the information in the solution found by IMINOS severely limits the possible actions of each task, frequently allowing no choices whatsoever, and in practice we have found the search to be quite fast. A global state contains the states of the DFAs for all the tasks and constraints (the behavior generator uses the DFA representation for all tasks and constraints, converting regular expressions to DFAs as necessary) as well as the symbol and arc counts being used to guide the search. These counts represent the remaining number of times a symbol or arc may occur; they are started at the values given by the solution and decremented to zero. Once at zero, a count prohibits its symbol or arc from being taken in any successor of that global state, pruning the search tree. The search starts at the global state in which all task and constraint DFAs are in their start state and all counts to be used are set to the value found by the solution. The global state space is searched depth first (to a user specified depth bound) until a *final global state* is found in which all task and constraint DFAs are in accepting states and all

counts are zero.

The search from a given global state proceeds by generating and exploring all possible successor states. A global state Y is a successor of global state X in the state space if there exists an event symbol a in the alphabet of a task T such that

1. in global state X, task T can make a transition t on a to a state s, s is the state of T in global state Y, and the states of other tasks are the same in Y as they are in X,

2. in global state X, each constraint C_i having a in its alphabet can make a transition on a to a state s_i, s_i is the state of C_i in global state Y, and the states of all constraints not having a in their alphabet are the same in Y as they are in X,

3. the counts for the number of times symbol a and transition t may occur is one less in global state Y than in global state X, all other counts are the same in state Y as they are in state X, and all counts in state Y are nonnegative.

For efficiency, the tool does not check whether more than one task can make a transition on a since the alphabets of task expressions produced by the current deriver are disjoint. Heuristics, some of which are specific to CEDL, control the order in which successor states are generated and can eliminate some states that cannot lead to a final global state.

If a final global state is found, the list of event symbols allowing the global state transitions to the final global state is a trace of the concurrent system. This string of symbols and a list of each task's actions are written to a file and the analyst may then stop or continue the search for other behaviors. If no behavior is found within the given depth bound, then the analyst may extend the depth bound and continue the search from the states along the "frontier" of the space (states at the depth bound). If a solution to the system of inequalities is provided, the state space will be finite (there can be no more symbols than given by the solution) and so failure to find a behavior string within the depth bound given by the number of events in the solution proves no string satisfying the solution exists. The behavior generator also has facilities to allow the analyst that use the tool more interactively by using only a part (possibly none) of the solution and by modifying the solution to require or prohibit certain event symbols from occuring in the behavior string. Note, however, that the size of the state space increases rapidly as the amount of information given to the behavior generator decreases.

The behavior generator is written in Common LISP and makes use of the data types and functions provided by the constraint eliminator. It also uses data structures from the inequality generator to interpret the solutions.

4 Experimental Results

As noted above, we believe that the assessment of the significance of the analysis methods implemented by this toolset must include the application of the tools to a variety of types and sizes of concurrent systems. We have therefore used the toolset to analyze a number of examples, and we report the results of several of these experiments here. We discuss the examples in somewhat more detail than is perhaps customary, but we feel strongly that the strengths and weaknesses of the constrained expression approach cannot be understood without examining the effects of sometimes subtle variations in the systems being analyzed. We have tried to present enough results to show the effect of various factors on performance of the constrained expression tools, though we do not claim to be able to assess the import of these factors independently. These factors include the number of tasks in the system, the complexity of dataflow in the tasks, and what seem to be superficial differences in coding style. Many of the examples have also been analyzed by other researchers using other analysis methods. The next section includes some comparisons between our results and theirs. The CEDL code for the examples discussed in this section is too long to include in this paper, but is available from the authors.

4.1 Dining philosophers

Perhaps the most widely known example in the concurrent systems literature is Dijkstra's dining philsophers. This system models a group of philosphers who periodically think and eat. The philosophers eat at a round table with one seat for each philosopher and one fork between each pair of seats. A philosopher requires two forks to eat and each philosopher who wants to eat attempts to pick up one fork, say the one on the left, and then the other, eats, and then puts the forks down. The system is interesting because of the possibility of deadlock caused by all the philosophers picking up the forks on their left, leaving each of them unable to pick up the second fork. Various approaches can be used to prevent the deadlock.

We have analyzed several variations of this system. In the basic one, described in Section 2.2, we model each fork by a task with two entries. Calls to the "up" entry represent the fork being picked up a philosopher and calls to the "down" entry represent the fork being put down. The fork task loops forever, accepting calls first at its up entry and then at its down entry. Each philosopher is represented by a task that repeatedly calls the up entry of the fork to its "left", the up entry of the fork to its "right", and then the down entries of the two forks. A system with n philosophers thus has $2n$ tasks. Analysis is intended to detect the possibility of deadlock.

In Figure 13, we show the performance of the constrained expression toolset on several sizes of the basic dining philosophers system, for all of which the toolset produced a system trace displaying deadlock. The columns give, respec-

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	40	109	4	26	9	21	381 × 320	169
40	80	203	11	77	71	46	761 × 640	408
60	120	298	21	158	74	78	1141 × 960	629
80	160	403	35	248	75	122	1521 × 1280	883
100	200	501	60	399	120	169	1901 × 1600	1249

Figure 13: Toolset Performance on Basic Dining Philosophers Problem

tively, the number of philosophers, the number of tasks in the system, the time in seconds used by the deriver, the eliminator, the inequality generator, IMINOS, and the behavior generator, the size of the system of inequalities (number of inequalities × number of variables), and the total time used by the toolset. All the experiments reported in this paper were run on a DECstation 3100 with 24 MB of memory; times given are in CPU seconds on that machine and include both user and system time.

All the IMINOS runs in this table used the sum of the variables corresponding to the operands of STAR operators in the task expressions as the objective function. We note that the performance of IMINOS on these examples is quite sensitive to the particular objective function used. When we used the sum of all the variables or a constant objective function, IMINOS reported that the system of inequalities for the 30-philosopher system (and all bigger ones) was inconsistent. The difficulty appears to be due to stability problems related to the bandedness of the system of inequalities. We discuss these issues further in Section 5.

One of the standard ways to prevent deadlock in the dining philosophers system is to introduce a "host" or "butler" who ensures that all the philosophers do not attempt to eat at the same time. We have modelled this by introducing an additional host task and modifying the philosospher tasks. The host task has two entries, "enter" and "leave", and a philosopher must rendezvous with the host at "enter" before attempting to pick up the first fork. After putting down the second fork, the philosopher calls the "leave" entry. The host keeps track of the number of philosophers in the dining room (the number of rendezvous that have occurred at "enter" minus the number at "leave") and repeatedly accepts calls at "enter" as long as no more than $n-2$ philosophers are in the dining room. The "leave" entry is unguarded, so calls at that entry can be accepted at any time.

Although the dining philosophers system with host and n philosophers involves only one more task than the basic system with the same number of philosophers, the sytems of inequalities and some of the tool execution times are much larger for the systems with host. Control flow in the system with host depends on the value of the variable counting the number of philosophers in the

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	41	140	105	157	65		603 × 1261	467
30	61	190	437	538	58		903 × 2491	1223
40	81	265	1079	1516	81		1203 × 4121	2941

Figure 14: Toolset Performance on the Dining Philosophers with Host

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	41	141	128	171	222	54	607 × 1305	716
30	61	196	392	537	296	119	905 × 2523	1540
40	81	259	1104	1603	865	239	1205 × 4163	4070

Figure 15: Toolset Performance on the Dining Philosophers with Erroneous Host

dining room. The constraint eliminator intersects the task expression for the host and the constraint involving this variable, so that the system of inequalities properly reflects the dependence of control flow on the number of philosophers in the dining room and the analysis does not spuriously report deadlock, but this process, together with the additional entry calls in the philosopher tasks, leads to significantly bigger systems of inequalities.

Again, we analyzed these systems to detect possible deadlock due to all the philosophers picking up forks. Performance of the toolset for n dining philosophers with host is shown in Figure 14 for several values of n. The columns in the table show the number of philosophers, the number of tasks, the times for the various tools, the size of the system of inqualities, and the total time, just as in Figure 13.

In each case, IMINOS reports that there is no integral solution to the system of inequalities, implying that no such deadlock is possible (in these cases as well, we used the sum of the variables corresponding to the operands of STARs as the objective function). It is therefore not necessary to run the behavior generator in these cases.

For comparison, we also analyzed the dining philosophers with host in the case where an incorrect guard on the host's "enter" entry allows all the philosophers to enter the dining room at once. The performance of the toolset on these problems is shown in Figure 15. In each case, the toolset produced a behavior exhibiting the deadlock.

Several other versions of the dining philosophers problem have been considered by other authors. We report briefly on the analysis of two of these with

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
5	11	59	2	10	3	10	144 × 174	84
5	6	49	83	57	6	18	303 × 585	213
5	6	51	29	6	1		73 × 95	87

Figure 16: Toolset Performance on Other Versions of the Dining Philosophers Problem

the constrained expression toolset.

Young, Taylor, Forester, and Brodbeck [36] used their CATS system to analyze three-, four-, and five-philosopher examples of an "unrolled" version of the dining philosophers with host. In this version, the host task does not use a variable to keep track of the number of philosophers in the dining room, but instead uses nested `select` statements. The CATS system was used to verify a temporal logic assertion (that, under the assumption of a fair scheduler, each philosopher can get into the dining room). We used the constrained expression toolset to analyze the five-philosopher system for deadlock. The design published in [36] is not equivalent to the one in which the host uses a variable to keep track of the number of philosophers in the room (as was pointed out to us by Sol Shatz), and the constrained expression toolset correctly detects a possible deadlock in the "unrolled" version.

Karam and Buhr [1] analyze several versions of the dining philosophers problem for deadlock and starvation. Their systems use a single fork manager task to model the forks, rather than individual tasks. We analyzed CEDL versions of two of these systems for deadlock.

Results for these systems are shown in Figure 16. The first row of the table gives the results for the five-philosopher "unrolled" system of [36]. In this case, the toolset produces a system trace displaying the deadlock. The second row gives results for a system with a fork manager in which deadlock is possible, and the third row gives results for a system in which the fork manager prevents deadlock by requiring the philosophers to pick up both forks at the same time. In the first of these cases, the toolset produces a system trace displaying deadlock. In the second, IMINOS reports that no deadlock is possible, and it is not necessary to run the behavior generator.

4.2 Gas station

The automated gas station example introduced by Helmbold and Luckham [7] has been studied by a number of authors (e.g., [1], [37]). This system models an automated gas station with an operator, a number of pumps, and a collection of customers. A customer pays the operator, who then activates a pump as appro-

priate. The customer then pumps gas, and the pump informs the operator of the amount pumped. The operator then gives change to the customer. Helmbold and Luckham used a series of iterative refinements of this system to illustrate their run-time monitoring system for debugging Ada tasking programs. Their examples involved systems with ten customers and three pumps.

We have analyzed several versions of the system that correspond to some of the refinements used by Helmbold and Luckham. In the first of our systems, there are two customer tasks, one pump task, and one operator task. Since we are interested in the concurrent aspects of the design, rather than the details of the computations performed by the various tasks, we ignore the amount of money paid by customers and the amount of change received. In this version, the operator does not activate the pump for a waiting customer until change has been given to the other customer. Because of a race between two customers who have both prepaid, the operator may attempt to give change to a customer who has not yet pumped gas, leading to deadlock. Our analysis is intended to detect this deadlock. In a second version, again with two customers, the operator activates the pump for any waiting customer before giving change. In this case, such a deadlock is impossible, and the toolset reports this. (Note that, even though deadlock is avoided, it is still possible for a customer to receive another customer's change. Karam and Buhr's [1] critical race assistant points up this possibility.)

When this deadlock-free two-customer design is scaled up to three customers, however, a more complicated race condition arises, again leading to the possiblity of deadlock. (This was first noticed by K. C. Tai [38], who used a graphical analysis method to detect the error.) We analyzed two versions of the three-customer extension of this problem. The first is a straightforward extension. In this case, the constraint eliminator produces an REDFA for the operator with a very large number of states due to the possible states of the queue of waiting customers. The large number of states (5,239 in the DFA produced by the eliminator, 433 in the corresponding REDFA) is responsible for the fact that the eliminator takes more than 30 minutes in this case. The number of states can be reduced by setting the variables corresponding to slots in the queue to some fixed value when that slot is not occupied by a customer waiting for service. (Since that practice would allow standard dataflow techniques to detect certain errors, it might be good programming style in general.) The toolset finds the deadlock in both of these versions of the gas station.

Results for these systems are shown in Figure 17. The first line gives results for the original two-customer gas station, in which deadlock occurs and the second line gives information for the revised two-customer version without deadlock. Results for the three-customer extension are shown in the third line of the table, and those for the version that reduces the number of states are given in the fourth line. The first column in the table shows the number of customers for each problem; the other columns are the same as in the preceding figures. We note that these systems have many fewer tasks than the dining

cus	tasks	deriv	elim	ineq	IMINOS	behav	size	total
2	4	36	26	11	5	8	120 × 200	86
2	4	37	30	12	5		125 × 209	84
3	5	44	2034	202	361	195	604 × 1401	2836
3	5	46	644	63	164	53	315 × 643	970

Figure 17: Toolset Performance on the Gas Station

cus	tasks	deriv	elim	ineq	IMINOS	behav	size	total
2	4	37	3	5	2	4	76 × 80	51
2	4	34	2	3	1		65 × 63	40
3	5	45	13	7	8		107 × 131	73

Figure 18: Toolset Performance on Gas Station with Separate Entries for the Customers

philosophers examples, but the systems of inequalities and the tool execution times are relatively large. This chiefly reflects the more complicated dataflow in the operator task.

One way to avoid deadlock and ensure that customers receive their own change is to have separate entries in the operator and pump tasks to distinguish the customers. In these systems, the operator task maintains a flag for each customer indicating whether that customer has prepaid and is waiting for change. The number of states in the REDFA for the operator task is much smaller than that in the versions discussed earlier. Our analysis was intended to determine whether a customer who has prepaid can be permanently blocked before pumping gas. The toolset correctly determines that this cannot occur in the versions with two and three customers. The sum of all variables was used as the objective function in these cases; performance with this objective function was much better than when the sum of variables corresponding to STAR operands was used.

For comparison, we also analyzed a version with two customers with an error (similar to that in the two-customer version discussed previously) that permits deadlock to occur. Results for the correct and incorrect versions of these systems are shown in Figure 18. The first row gives results for the erroneous two-customer version. The next two rows give the results for the correct versions. It is not necessary to use the behavior generator in the latter cases.

4.3 Readers and writers

Another standard example from the concurrent systems literature is the readers and writers problem. In this problem, readers and writers attempt to gain access to a shared resource. Readers can share access, but the resource can be corrupted if more than one writer gains access at the same time and readers may get inconsistent data if a writer and one or more readers use the resource simultaneously. Various versions of this problem have been considered, with priority schemes and other variations. We analyzed some CEDL versions of the problem for deadlock and to determine whether a writer and one or more readers could gain access to the resource at the same time.

These systems consist of a number of tasks representing readers and writers, and a controller task that the others call in order to gain access to the resource. The analysis for deadlock is similar to the analyses described above. The analysis for simultaneous access by readers and writers is quite different, and requires some discussion.

A reader gains access to the resource through a rendezvous with the controller at its START_READ entry, and relinquishes access through a rendezvous at END_READ. Similarly, a writer gains and relinquishes accesses through rendezvous at the entries START_WRITE and END_WRITE. Simultaneous access by a reader and a writer would thus be represented in a system trace by an occurrence of a symbol representing the rendezvous at START_WRITE between symbols representing corresponding rendezvous at START_READ and END_READ, or by the occurrence of a symbol representing a rendezvous at START_READ between symbols representing corresponding rendezvous at START_WRITE and END_WRITE. Detecting such simultaneous access in a system trace depends on determining that symbols occur in that trace in a particular order, and the inequalities we generate do not reflect the order of symbol occurrences. For this reason, our toolset cannot directly address this question. In order to analyze the readers and writers system for undesirable simultaneous access to the resource, we therefore modified the controller task so that, at each START_READ or START_WRITE rendezvous, it checks to determine whether a reader and a writer both have access to the resource and sets a flag if this is the case. Our analysis then asks whether the symbol representing the setting of this flag occurs in any trace of the system.

Results for a few versions of these readers and writers systems are shown in Figure 19. The first column of the table contains an ordered pair giving the number of readers and the number of writers in the system. The first line of the table gives times for an incorrect system with four readers and one writer. In this system, an error in the controller task allows a deadlock. The second line gives results for a correct system that is analyzed for undesirable simultaneous access to the resource. In this case, the constraint eliminator removes that part of the controller task expression containing the symbol representing the setting of the flag, and it is not even necessary to generate a system of inequalities to determine that the flag is never set. The time shown for the inequality generator

(r,w)	tasks	deriv	elim	ineq	IMINOS	behav	size	total
(4,1)	6	40	6	6	3	3	82 × 137	58
(4,1)	6	40	5	2				47
(4,1)	6	41	7	7	4		90 × 148	59

Figure 19: Toolset Performance on Readers and Writers Problem

in the table is just the time required to determine that the symbol does not occur in the constrained expression produced by the constraint eliminator. The third line gives results for a system in which the controller gives the writer priority by accepting a call at START_WRITE at any time, but then disabling the entry START_READ and waiting for all readers who have access to the resource to relinquish it before allowing the writer to proceed. This system, which is correct, was analyzed to detect deadlock.

4.4 Distributed mutual exclusion

We now describe some experiments with a system for achieving mutually exclusive use of a resource in a distributed system.

The system analyzed is a CEDL version of a design that implements part of an algorithm for mutual exclusion due to Ricart and Agrawala [39]. In that algorithm, a node wishing to obtain exclusive use of the resource sends a request to each of the other nodes in the system, and then waits for a reply from each node before proceeding to use the resource. A node receiving a request decides whether to reply immediately, thereby granting its permission to use the resource, or to defer its reply until it has used the resource itself. This decision is determined in part by a sequence number sent as one portion of the request message and in part by a fixed priority ordering on the nodes that is used in case two sequence numbers are equal. The sequence numbers are generated by the individual nodes and are similar to the numbers used in Lamport's "bakery algorithm" [40].

The constrained expression approach was applied in [41] to detect an error in a partial design for a system implementing the Ricart-Agrawala algorithm, and then to show that the error was eliminated in a modified version of the design. In that paper, the design was written in DYMOL, a language with asynchronous message passing. We have used the toolset to examine a similar design written in CEDL.

In the CEDL version, each node of the distributed system consists of two tasks: an "invoker" that generates requests to use the resource and a "request handler" that responds to requests from other nodes. When the invoker decides (nondeterministically) to use the resource, it sends requests to the other nodes and informs its request handler, so that the request handler can determine

whether to defer requests arriving from other nodes. When the invoker has received replies from the other nodes, it uses the resource, and then notifies its request handler that it has done so. We begin by considering a preliminary design for a single node, in which the details of the Ricart-Agrawala algorithm have not yet been elaborated. In this design, the request handler task always defers requests from other nodes when its invoker has indicated its desire to use the shared resource. Two other nodes in the system are simply stubs that generate requests to use the resource and replies to requests from the invoker of the elaborated node. The analysis is intended to determine whether a request received at the elaborated node may be permanently deferred. The toolset shows that this cannot happen. This is essentially equivalent to the analysis performed by hand in [41], though the different communication primitives in CEDL and DYMOL make the details of the designs quite different.

We next consider two versions of a system with three elaborated nodes and an additional task simulating the resource. In this case, we want to detect possible violation of mutually exclusive use of the resource. As in the readers and writers examples discussed above, the resource task sets a flag if two invokers use the resource simultaneously, and the query the toolset attempts to answer is whether that flag is ever set in a behavior of the system. Note that deadlock is possible in this system because the full algorithm used by the request handler tasks to determine when to defer requests has not yet been implemented at this stage of the design process, and all the nodes could decide to defer each other's requests.. But a correct design at this stage should enforce mutually exclusive use of the resource.

In the first of these examples, the invoker task in each node sends requests to the other nodes before notifying its own request handler of its intention to use the resource. It is thus possible that a request handler could already have given permission for another node to use the resource. IMINOS finds a solution to the system of inequalities, but, due to the problem with cycles in the REDFA discussed in section 3.3, this solution is spurious. We then manually add the linear inequalities necessary to exclude solutions that incorrectly give nonzero values for arcs in those cycles, as described in that section, and run IMINOS again. Again the behavior generator reports that the solution is spurious. Examination of the output of the behavior generator shows that, in the course of trying to construct a system trace corresponding to the solution, the behavior generator reached a global state in which all the tasks are blocked, but no replies have been deferred. We have thus detected a possible deadlock of the three-node system due to the error in the invoker tasks, rather than the deferral of requests.

In the second version of this three-node system, the invoker correctly notifies its own request handler before requesting permission from the other nodes, so that the resource is used in a mutually exclusive fashion. In this case as well, the problem with cycles leads to a solution that does not correspond to a behavior, and we manually add inequalities as before. IMINOS finds a solution to the new

tasks	deriv	elim	ineq	IMINOS	behav	size	total
6	46	11	4	3		129 × 186	70
7	87	30	8	71	35	216 × 247	238
7	85	30	8	72	60	216 × 247	262

Figure 20: Toolset Performance on the Distributed Mutual Exclusion Examples

system of inequalities, but the behavior generator correctly reports that this solution is also spurious. The solution found by IMINOS reflects a "behavior" in which the events occur out of order — each of two request handlers behaves as if a request from the other node was received before a rendezvous with its own invoker, but the invokers rendezvous with their own request handlers before sending the requests to the other nodes. The problem here is that the system of inequalities produced by the inequality generator does not fully reflect the order in which the corresponding events occur. At this time, we do not know of a general method for solving this problem, which, as in this case, can lead to spurious solutions. The behavior generator can tell us that this particular solution does not correspond to a behavior, but, in cases like this one, the toolset does not give a definitive answer to the question of whether there is a behavior with the property the analyst is interested in.

Times for the three examples discussed in this subsection are shown in Figure 20. The first row gives the results for the analysis of the system with a single elaborated node. The second row gives the results for the incorrect system with three elaborated nodes, and the third row gives the results for the correct version of that system. In the second and third rows, the times shown for IMINOS are for the second run, in which additional inequalities have been added. These times are somewhat longer than for the runs without the additional inequalities.

5 Assessing the Constrained Expression Toolset

At the beginning of this paper, we argued that an assessment of the value of a method for analyzing concurrent software must necessarily include an empirical evaluation of the application of that method to a variety of types and sizes of concurrent systems. The constrained expression toolset we have described was constructed with the intention of conducting such an empirical evaluation, and we have presented some of the results of our initial efforts in that direction. In this section, we consider various aspects of that evaluation and discuss our current assessment of the constrained expression approach. We then briefly compare it to some related methods.

5.1 Performance and scalability

As the results described in the previous section illustrate, the constrained expression toolset is capable of analyzing large systems. The toolset carries out a complete analysis of the basic dining philosophers problem with 100 philosopher tasks and 100 fork tasks, starting from the CEDL code and producing a behavior displaying deadlock, in less than 21 minutes. When the behavior of the individual tasks is more complex, the toolset cannot handle quite so many tasks, but it is clear that it can be used with at least some systems that approach, or even exceed, realistic sizes for concurrent system designs. This is in marked contrast to the results reported for most other methods that have been implemented, notably those based on constructing and searching a reachability tree. This ability to analyze large systems is the most obvious strength of the approach.

In fact, with a very slight modification, the toolset can be used to analyze systems that include an extremely large number of identical tasks. If there are n identical tasks in the system, we can simply set the variable corresponding to the root node of the parse-tree of the task expression (or to the flow into the inital state of a task DFA or REDFA) to n, rather than one. This corresponds to starting n identical copies of the task with that task expression. In conjunction with this technique, we have also experimented with the use of an integer programming variable to represent a CEDL variable used by a task in the system to maintain a count of some sort. At this time, the latter technique can only be used with certain types of systems, and the behavior generator will need some modification for use with these two techniques, but we present in Figure 21 some results of applying the other components of the toolset to a system involving two coupled resource managers controlling equal amounts of two resources and a large number of identical customers who require both resources.

The figure shows the number of customer tasks, the amount of the first resource originally available, the amount of the second resource originally available, the number of tasks in the systems, and the times used by the components of the toolset. The analysis is intended to detect the possibility that the controller of the second resource grants more requests for access to the resource than can be accommodated by the available amount. The first two lines of the table give the results for systems with 400 customers; the first line shows a correct system and the second shows one with fewer units of the second resource, leading to an error. The third and fourth lines give the results for similar systems with 800 customer tasks. Because the variables used to count resource units in the two controllers are represented by integer programming variables, it is not necessary to use the constraint eliminator in these analyses. The solutions found by IMINOS for the two incorrect examples do indeed correspond to system traces displaying the pathological behavior. Note that the systems of inequalities are the same size and the execution times are the same for all versions of the system. Ongoing research involving these techniques is discussed

cus	r1	r2	tasks	deriv	ineq	IMINOS	size	total
400	380	380	402	25	3	2	36 × 39	30
400	380	379	402	25	3	2	36 × 39	30
800	780	780	802	25	3	2	36 × 39	30
800	780	779	802	25	3	2	36 × 39	30

Figure 21: Toolset Performance with Many Identical Tasks

briefly in Section 6.

Problems in the performance of the integer programming component of the toolset do arise with large systems, however, and raise some serious issues concerning use of the toolset. Particularly significant is the fact that the results obtained by IMINOS are sensitive to the objective function chosen, and indeed are incorrect for large versions of the basic dining philosophers problem with one objective function we have examined. This appears to be due to numerical stability problems that arise here from an interaction between the particular objective function and the bandedness of the coefficient matrix. This bandedness reflects the communication structure of the concurrent system — each task communicates only with two "nearby" tasks — and is known to cause difficulties for the particular simplex algorithm used in MINOS, but we do not understand the problem well enough at this time to be able to predict accurately the cases in which it will arise. In other cases, notably those with complex dataflow, the presence of many solutions to the LP relaxation of our integer programming problem when there is no integer solution leads to extremely long run times for IMINOS. There appears to be significant potential for improving the performance of the integer programming component of our toolset by modifying the branching algorithm used by IMINOS and possibly also by implementing other approaches to integer linear programming that might take better advantage of special characteristics of our systems of inequalities. Some of these possiblities are discussed in the next section.

The performance of the toolset is not easily predicted from known results on the computational complexity of the algorithms it implements. The translation process implemented by the deriver is essentially linear in the number of tasks and the size of each task. In general, the "intersection" of DFAs performed by the constraint eliminator increases the sizes of the state spaces exponentially, but our eliminator needs to do this only a small number of times and its performance is adequate for our analysis. The complexity of inequality generation, like the translation process carried out by the deriver, is certainly linear in the size of the constrained expression, which could in principle be exponential in the size of the original concurrent system. Integer linear programming is known to be *NP*-complete (the satisfiability problem can be formulated as a 0-1 integer linear programming problem), and the worst-case performance of any branch-

and-bound algorithm is exponential in the size of the coefficient matrix. The average-case performance for the algorithm we have implemented is not known, however, and the performance of IMINOS does appear to be the limiting factor in our ability to handle several of the examples. Finally, the search carried out by the behavior generator is clearly exponential in the number of tasks in the system in general, but frequently is severely constrained by the solution found by IMINOS. For instance, the behavior generator does no backtracking in the dining philosophers problem with 100 philosophers.

The ability of the toolset to handle large problems is not obvious from theoretical investigation. We feel that this strongly supports our assertion that empirical evaluation is a necessary component of the assessment of analysis methods.

5.2 Range of problems that can be analyzed

The constrained expression toolset can be used to answer several of the most important types of questions developers of concurrent systems are likely to ask. The results presented in Section 4 show how the toolset can be used to answer questions about deadlock and violation of mutual exclusion. We have also used the toolset to detect blocking of single processes. In [42], we have shown how the toolset can be extended to answer questions about the timing properties of a concurrent system.

The current version of the constrained expression toolset, however, is not able to address questions about fairness or starvation. These questions involve infinite behaviors, and the constrained expression formalism does not describe infinite behaviors. Although an argument might be made that infinite behaviors are of little relevance to real computer systems, they provide an elegant way to formulate questions that certainly are relevant and must be addressed by the designers of such systems.

In addition, many questions about the order in which events occur can be answered by the toolset only if they can be translated into ones involving the number of occurrences of events. While this can often be accomplished by slightly modifying the system being analyzed, as in the readers/writers example reported in the previous section, such modifications represent an extra complication and are not always practical.

The toolset does correctly represent the dependence of control flow on intra-task dataflow. Some reachability-based methods intentionally ignore information about the values of variables in order to reduce the number of states that must be generated and examined. For example, the version of the CATS suite of tools described in [36] is unable to determine that deadlock is impossible in the dining philosophers with host for this reason. (Other reachability-based methods, such as [1], do correctly deal with dataflow.)

However, the ability of the toolset to analyze systems having tasks with very complex dataflow is limited. The problem, as for the reachability-based

methods, is the explosion in the number of states that must be considered. Furthermore, the toolset does not use information about the dependence of control flow on data when that information involves several tasks. For example, it can be seen from the customer and pump tasks in the gas station examples that the customer who has just called the PUMP.START_PUMPING entry must be the one who next calls the PUMP.FINISH_PUMPING entry, but our toolset does not use this fact to simplify the task expression for the pump. We are currently investigating some ways to make better use of this sort of information.

The integer programming component of the toolset sometimes produces "spurious" solutions to the systems of inequalities, that is, solutions that do not correspond to behaviors of the concurrent system. This is due to the fact that our systems of inequalities do not fully reflect the semantics of constrained expressions. The inequalities we generate do not directly restrict the values of variables corresponding to STAR operands in task expressions or arcs in cycles in task DFAs or REDFAs, and are unable to guarantee consistent ordering of events in different tasks because they involve only the total number of times an event occurs or an arc in a DFA is traversed. As demonstrated in the experiments with the distributed mutual exclusion system, it is sometimes possible to deal with spurious solutions arising from STARs or cycles in an ad hoc manner. At the present time, we are not able to eliminate spurious solutions due to problems with the order of occurrence of events, although the behavior generator does tell us that the particular solution found by IMINOS does not correspond to a trace of the concurrent system. Of course, even when the behavior generator reports that a solution of the system of inequalities does not correspond to a trace, it is possible that some other solution does correspond to a trace. Our analysis in the case in which the solution found by IMINOS does not correspond to a trace is therefore not conclusive.

The problems with spurious solutions due to STARs and cycles depend to some degree on the "coding style" of the example. We have found, for example, that such spurious solutions can often be prevented by guarding all entries as strictly as possible. In some cases, much stronger guards are possible in certain versions of a design than in others, although the versions appear essentially equivalent to most programmers. For instance, in the versions of the gas station in which customers call separate entries to identify themselves, we guard those entries with flags that indicate whether the customer has prepaid but not yet received change. Thus, the operator will accept a call at the entry through which a customer prepays only if the flag indicating that this customer has already prepaid is not set. Although it may be good coding style to include such guards as a defense against errors in other parts of the program, they are not necessary for its correct execution since customers do not attempt to prepay again before receiving change. In a version of the gas station in which customers identify themselves by passing a parameter during the rendezvous, however, all customers use a single entry to prepay and it is not possible to guard this entry as tightly since the operator cannot know in advance who will

call it next. In this version, we get spurious solutions due to the presence of cycles in the REDFA for the operator.

Another aspect of coding style that affects analysis is illustrated by the two three-customer versions of the gas station in which the operator maintains a queue of waiting customers. As shown in Figure 17, the version in which the variables representing slots in the queue are set to a fixed value when not in use has approximately half as many inequalities and integer programming variables and takes substantially less time to analyze than the version in which the variables are not reset. Indicating that the programmer is no longer interested in particular variables at a certain point in the program might be good practice for other reasons as well; standard dataflow analysis techniques, for example, could report later use of these variables without intervening assignments. In fact, the process of detecting such variables and resetting them to some value would be relatively easy to automate using such dataflow analysis techniques. We have not attempted to incorporate such automated resetting of variables into our toolset as yet.

5.3 Comparison with other methods

We now briefly compare the constrained expression toolset and the analysis techniques it implements with some related approaches.

Several investigators have implemented analysis techniques for concurrent systems based on generating and examining some sort of reachability graph for states of the system (e.g., [1], [2], [36]). In general, the number of states such methods must examine is exponential in the number of tasks in the system [28], and different approaches are taken to reducing this complexity. For example, the CATS system uses "task interaction graphs" and ignores the values of variables in order to reduce the number of states, while the starvation and critical race analyzers described by Karam and Buhr [1] work from a temporal logic specification. Similarly, the Petri net reduction techniques of [37] are intended to reduce the size of a Petri net representation of a concurrent Ada program in order to make reachability analysis practical.

It appears that none of these techniques can currently deal with systems as large as some of those analyzed using the constrained expression toolset. For example, Karam and Buhr 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, Taylor, Forester and Brodbeck suggest that a reasonable granularity for analysis of designs is "in the neighborhood of 8 processes."

These reachability-based methods, however, can be used to answer questions that cannot be addressed by the constrained expression toolset. Both the CATS system and Karam and Buhr's starvation analyzer can be used to verify temporal logic assertions involving such questions as fairness, as well as detecting deadlock. And with small systems, reachability-based analysis can be quite effi-

cient. The times reported by Karam and Buhr for analysis of the two-customer gas station, for example, are significantly lower than the corresponding times for the constrained expression toolset. (Karam and Buhr begin with a logical specification, rather than standard source code, and so do not report times for tools corresponding to our deriver.)

In some cases, the size of the reachability graph that must be generated can be sharply reduced. McDowell [43], for example, has described a method for collapsing parts of the reachability graph when the system includes a large number of identical tasks. (This is the case in which we have experimented with setting a varable to n, rather than one, as discussed above.) Valmari [44] has described a method that can detect deadlock in systems with communication structure like that of the basic dining philosophers in time that is linear in the number of tasks. The range of useful application of this method is unclear at the present time — for the dining philosophers with host, for example, the method remains exponential in the number of tasks — but this approach is the only one we know of other than constrained expressions that can handle systems with more than 100 tasks.

Another approach, very closely related to ours, is the Petri net invariant method of Murata, Shenker, and Shatz [45]. In this method, certain Petri nets are derived from Ada tasking programs, and the T-invariants of these nets are determined. The T-invariants are integer solutions to a homogeneous system of linear equations and correspond to counts of transition firings whose net effect is to return the derived Petri net to its original marking (representing a deadlock-free execution of the original Ada program). Some T-invariants correspond to possible firing sequences of the net, but others do not, essentially because the process of finding T-invariants ignores the restrictions on the order in which transitions can fire that are imposed by the semantics of Petri nets. These "spurious" T-invariants are thus similar to the solutions of our systems of inequalities that do not correspond to traces of CEDL systems.

The approach of [45] is to use the T-invariants first to detect and remove certain "inconsistency" deadlocks, and then to guide the construction of a reachability graph to determine whether "circular" deadlocks are possible. This approach is very similar to that implemented by our toolset. A major difference is that, in the method of [45], a T-invariant is a solution to equations representing necessary conditions that must be satisfied by transition counts corresponding to a correct execution of the Ada program. In our approach, the inequalities represent necessary conditions that must be satisfied by a behavior of the CEDL system satisfying the property corresponding to the analyst's query, and we have usually used the method to detect pathological behaviors. Also, we use a solution to the inequalities to guide a reachability-based search for a corresponding behavior, while Murata, Shenker, and Shatz use the T-invariants to guide search at a later stage of their analysis.

6 Other Directions and Future Research

In this section, we briefly describe several directions for ongoing and planned research on extending the constrained expression approach and analysis techniques.

6.1 Extending and improving the toolset

We have planned a number of changes in the toolset intended to improve its performance or provide new functionality. Work on implementing these changes has already begun.

We have begun modifications to the toolset to support more fully the analysis of systems with large numbers of identical tasks described previously. These modifications will affect most of the components of the toolset to some degree: for example, the task expression of the task with multiple copies must be identified to the deriver and the other tools, the number of copies of the task must be specified, the inequality generator must set the appropriate variables to n rather than one, and the behavior generator must be modified slightly to deal with the solutions to these systems of inequalities. In the experiments described above, the necessary changes in the system of inequalities were made simply by editing the input file for IMINOS, but we have already begun to make the necessary modifications to automate this kind of analysis. The changes are all relatively minor. Similar modifications will be necessary to automate the use of an integer programming variable to represent CEDL variables used as counts.

We plan a number of changes in the implementation of the deriver that will increase its speed and remove some minor restrictions on CEDL programs, such as the prohibition on global variables and the requirement that variable and entry names be unique. A major part of these changes will be based on replacing the semantic analyzer currently used by the deriver with a new one currently under development as part of the Arcadia project. These changes will not make a large difference in the performance of the toolset, but they will make it easier and more convenient to use.

We noted above that the present implementation of IMINOS uses a naive strategy to choose a variable on which to branch. We are planning to replace this with a strategy based on the *special ordered sets* first introduced by Beale and Tomlin [46]. These are sets of variables of which exactly one can be nonzero. In our case, the variables representing the ways in which a particular task can stop executing (normally or abnormally) form such a set, and each of these variables takes values only in the range from zero to one even in solutions to the LP relaxations. Branching first on variables from these sets should significantly reduce the search required to find an integer solution to our inequalities and reduce the possibility of cycling in the choice of branching variables.

We are also investigating other approaches to solving our systems of inequalities. The basic branch-and-bound approach does not take advantage of the fact

that a significant part of our system of inequalities is totally unimodular (essentially a network flow problem). The use of a special-purpose algorithm, such as the Lagrangian relaxation approach of [47], that makes use of this structure might lead to very large improvements in the performance of the integer programming component of our toolset. Because selecting and implementing the right algorithm will require a major effort, we view this as a longer term project.

We have a number of ideas for improving the behavior generator as well. For example, it will be relatively easy to modify the behavior generator so that, when a solution found by IMINOS is spurious due to problems with a cycle in a task DFA, the behavior generator informs the analyst and assists in adding the inequalities that eliminate such solutions. We also intend to investigate improvements in the heuristics it uses in its search.

Finally, we plan improvements in the interfaces between the various tools and between the user and the toolset. Improvements in the interfaces between the tools will allow us, for example, to save more of the intermediate results of analysis and reduce some duplication of effort by the tools. The current user interface is extremely rudimentary, and improvements in it will greatly increase the ease and convenience of using the toolset.

6.2 Extensions to the constrained expression analysis techniques

We also plan to extend the constrained expression formalism and analysis techniques to allow the toolset to be used with a wider range of problems and queries. Among the topics we are investigating are methods for directly handling more complex queries, such as "Can event a occur between events b and c?", ways to express infinite behaviors so that questions of fairness and starvation can be addressed, and ways to modularize the constrained expression representations of systems and their analysis. We sketch a few of these ideas below.

Because the inequalities we generate from a constrained expression only involve the total number of times something happens in a trace of the system, our toolset cannot, in general, directly address questions about the order in which events occur. (Of course, the behavior generator could be used directly to search for a system trace in which events occurred in the specified order, but, without a solution to the system of inequalities to reduce the search space, this is unlikely to be a practical approach.) Certain questions about order, however, are important in analyzing concurrent software. For example, questions about whether a resource is used in a mutually exclusive fashion by the tasks in a concurrent system are fundamentally questions about order. As the discussion of the readers/writers examples in Section 4.3 shows, such questions can sometimes be converted into ones that our toolset can address, and, indeed, answer efficiently. However, it would be better if such questions could be dealt with directly, and we are investigating an approach that seems quite promising.

A query such as "Can event a occur between events b and c?" requires consideration of the set of segments of system traces beginning with the event symbol corresponding to b and ending with the symbol corresponding to c. The idea is to add additional variables representing the possible beginning or end of a segment of a behavior at appropriate states in the task DFAs, and to generate inequalities from task DFAs as usual, but allowing flow in and out with these new variables. Adding inequalities stating that the total flow into and out of a DFA is one gives a system of inequalities describing segments of system traces. Combining two such systems would allow the toolset to answer queries about the occurrence of one event between two others, as well as other, more complex questions. We have not yet implemented this approach, but it is closely related to the method we have implemented for the analysis of real-time software, as described below.

As we noted above, the constrained expression formalism does not represent infinite system behaviors of the type required for consideration of fairness and starvation. We are investigating modifications to the formalism that would allow us to represent such behaviors and answer these sorts of questions using the toolset. The approach we are considering at this time involves the use of special symbols to denote the infinitely repeated occurrence of certain events. Special inequalities must be generated for such symbols, of course, to ensure that rendezvous counts match up properly between different tasks, but the general approach to analysis with these symbols would be essentially the same as that used for finite behaviors with the current formalism.

The current toolset was designed for the analysis of "closed" system designs, which describe complete, self-contained systems. (While CEDL allows incompleteness at the level of individual expressions and statements within tasks, it does not allow incompleteness at the level of tasks.) Real systems, however, are seldom self-contained, but must interact with an external environment. Moreover, hierarchical development methodologies, where a complex software system is organized and developed as a set of interacting (sub)systems, which are themselves organized and developed as sets of interacting (sub)systems, and so on, have become an accepted means for managing the complexity of software. Typical designs produced during the development of an actual software system, therefore, are neither self-contained nor complete, and we are currently investigating methods for representing and analyzing more general designs, which need not describe closed systems.

One straightforward method for accomplishing this was mentioned above. For analysis of a single node (subsystem) of the Ricart-Agrawala distributed mutual exclusion algorithm we introduced tasks to act as stubs that generate requests to use the resource and replies to the elaborated node's requests. In addition to providing a closed system for input to the deriver, the stubs used for this particular example guarantee that the elaborated node eventually receives a reply for every request that it makes. Thus, the stubs were also used to express an assumption about the environment in which the system (the elaborated node)

executes that was required for the analysis.

We are currently investigating ways that do not require the introduction of stubs to perform analysis of designs for systems that are not necessarily closed. For a CEDL design, the general idea is to derive a constrained expression representation from the design as if it describes a closed system, but omitting the rendezvous constraints for entries representing the interface between the system and its environment (i.e., entries of the elaborated system that are called from outside the system and entries that are called but not declared by the system), and to introduce *environment constraints* that express assumptions about the environment required for the analysis. Environment constraints describe restrictions imposed by the environment on the permissible patterns of symbols representing communication through the system's interface. Consider, for example, the design for a single fork task in one of the dining philosopher programs. The "up" and "down" entries define the interface for this "single-fork system". Environment constraints required for showing the single-fork system enforces mutual exclusion on the use of the fork stipulate that the philosophers to the right and left of the fork alternate in calling the up and down entries (beginning with a call to the up entry). We are also investigating a method for composing constrained expressions that will support modular analysis of systems. This method makes use of the features of the more general interpretation of constrained expressions introduced in this paper.

6.3 Analysis of real-time software

We have recently developed and begun experimenting with an extension of the constrained expression analysis techniques that can be used to assess the timing properties of concurrent systems, and have extended the toolset to implement this technique. We give here a short survey of the current status of this work and some additional ideas in this area that we are currently pursuing. Details of our method and an example of its application can be found in [42].

Applying the constrained expression analysis techniques to real-time systems requires extending the formalism to account for time. The most straightforward way to do this is to assign a duration to each event, so that the time required for a sequence of events is just the sum of the durations of the individual events in the sequence. However, this interpretation only makes sense when the events are non-overlapping, as would be the case if the concurrent system being analyzed were to be run on a single processor. We have adopted this extension to the formalism, and are currently investigating further extensions that would allow the analysis of "truly concurrent" (e.g., multiprocessor) real-time systems.

Our analysis produces an upper bound on the time that can elapse between the occurrence of two specified events by finding an upper bound on the duration of any subsequence of a system trace that begins with the first event and ends with the second. The basic idea is to generate systems of inequalities representing necessary conditions that must be satisfied by any such subsequence

of a trace, and to use IMINOS to find the maximum duration associated with any solution of the system.

As in the related method for dealing with queries involving order of events that we discussed above, the idea is to generate additional variables associated with the states in which the various task DFAs could be at the beginning and end of such a subsequence. We then generate systems of inequalities as usual, but using these new variables as possible flows into and out of the task DFAs.

The same factors that lead to spurious solutions of our systems of inequalities in the logical analysis discussed in this paper can lead to the upper bounds we obtain not being sharp. In particular, the presence of cycles in the task DFAs can result in IMINOS reporting that the duration is unbounded. We have developed a marking algorithm, described in [42], that greatly reduces the significance of this problem by removing arcs from the DFAs that cannot be reached in segments starting and ending with the specified events. In practice, we have found that our bounds are usually quite good, and frequently are indeed attained by subsequences of behaviors. We therefore believe that this kind of automated analysis can be of significant use in the development of real-time systems.

7 Conclusion

The constrained expression approach to analysis of concurrent software systems has several attractive features. It can be used with a variety of different design notations and programming languages that are based on different views of the semantics of concurrent computation, use different communication primitives, and are suitable for different stages of the development process. Developers of concurrent systems can thus use the notations and languages most appropriate for their tasks, while retaining the capability of rigorous analysis of their systems. Problems with combinatorial explosion are reduced, because analysis based on the constrained expression formalism does not require enumeration of a complete set of reachable states of the concurrent system. In addition, important aspects of the approach seemed relatively easy to automate.

Experiments with manual application of the constrained expression analysis techniques to small examples were quite encouraging. However, a determination of whether the techniques could really be of value to software developers could not be made without carrying out an empirical evaluation of their application to a wider range of examples, including examples far too large to analyze by hand. We therefore began to construct a toolset automating the main constrained expression analysis techniques. This paper describes that toolset and the analysis techniques it implements, and reports on our experiments with it.

The results of these experiments, as described in Section 4 indicate that the constrained expression toolset can be used to analyze systems that approach, and in some cases actually exceed, realistic sizes for concurrent system designs.

The toolset carries out a completely automated analysis, starting from source code in a design language and producing system traces displaying the properties represented by the analyst's queries, in many of these cases. Unlike several other approaches, it is able to deal with these large systems while retaining information about the dependence of control flow on the values of variables local to the components of the concurrent system. In its current form, however, the toolset is not capable of directly addressing certain questions about the behavior of concurrent systems. These include questions involving infinite executions of the system, such as starvation and fairness, and certain questions about the order in which events occur in executions. Our experiments have also pointed up certain other areas in which modifications to the toolset could significantly improve its performance.

The results of these experiments indicate the potential value of the constrained expression approach and certainly justify its continued development. Ongoing and planned research is directed at many of the issues identified by our experiments, as described in the previous section. This research involves improvements in the toolset to enhance its performance and make it easier and more convenient to use, and extensions to the constrained expression formalism and the analysis techniques automated by the toolset to expand the range of questions it can answer and concurrent systems it can analyze. Based on the results of the experiments conducted with the current version of the toolset and the improvements to be expected in the near future, we believe that the constrained expression approach can serve as a foundation for practical tools for developers of concurrent software.

References

[1] G. M. Karam and R. J. Buhr, "Starvation and critical race analyzers for Ada," *IEEE Trans. Software Engineering*, vol. 16, no. 8, pp. 829–843, 1990.

[2] S. M. Shatz and W. K. Cheng, "A Petri net framework for automated static analysis of Ada tasking behavior," *Journal of Systems and Software*, vol. 8, pp. 343–359, 1988.

[3] R. N. Taylor, "A general-purpose algorithm for analyzing concurrent programs," *Communications ACM*, vol. 26, pp. 362–376, May 1983.

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

[5] L. K. Dillon, "Verifying general safety properties of Ada tasking programs," *IEEE Trans. Software Engineering*, vol. 16, no. 1, pp. 51–63, 1990.

[6] S. Owicki and D. Gries, "An axiomatic proof technique for parallel programs," *Acta Informatica*, vol. 6, no. 4, pp. 319–340, 1976.

[7] D. Helmbold and D. Luckham, "Debugging Ada tasking programs," *IEEE Software*, vol. 2, pp. 47–57, March 1985.

[8] D. S. Rosenblum and D. C. Luckham, "Testing the correctness of tasking supervisors with TSL specifications," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification* (R. A. Kemmerer, ed.), pp. 187–196, 1989. Appeared as *Software Engineering Notes*, 14(8).

[9] G. S. Avrunin, L. K. Dillon, and J. C. Wileden, "Experiments with automated constrained expression analysis of concurrent software systems," in *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification* (R. A. Kemmerer, ed.), pp. 124–130, December 1989. Appeared as *Software Engineering Notes*, 14(8).

[10] 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. Software Engineering*, vol. 12, no. 2, pp. 278–292, 1986.

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

[12] L. K. Dillon, G. S. Avrunin, and J. C. Wileden, "Constrained expressions: Toward broad applicability of analysis methods for distributed software systems," *ACM Trans. Programming Languages and Systems*, vol. 10, pp. 374–402, July 1988.

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

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

[15] V. Pratt, "Modeling concurrency with partial orders," *International Journal of Parallel Programming*, vol. 15, no. 1, pp. 33–71, 1986.

[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications ACM*, vol. 21, pp. 558–565, July 1978.

[17] L. Lamport, *Paradigms for Distributed Systems: Computing Global States*, pp. 454–468. Lecture Notes in Computer Science 190, Springer-Verlag, 1985.

[18] M. K. Chandy and J. Misra, *Parallel Program Design.* Addison-Wesley, 1988.

[19] S. Katz and D. Peled, "An interleaving set temporal logic," in *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing*, pp. 178–190, 1987.

[20] S. Avery, "A tool for producing constrained expression representations of CEDL designs," Software Development Laboratory Memo 89-2, Department of Computer and Information Science, University of Massachusetts, 1989.

[21] L. A. Clarke, J. C. Wileden, and A. L. Wolf, "Nesting in Ada programs is for the birds," in *Proceedings of an ACM-SIGPLAN Symposium on the Ada Programming Language*, pp. 139–145, 1980. Appeared as SIGPLAN Notices 15(11).

[22] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers.* Reading, MA: Addison-Wesley, 1986.

[23] L. K. Dillon, "Overview of the constrained expression design language," Tech. Rep. TRCS86-21, Department of Computer Science, University of California, Santa Barbara, October 1986.

[24] R. N. Taylor, F. C. Belz, L. A. Clarke, L. J. Osterweil, R. W. Selby, J. C. Wileden, A. L. Wolf, and M. Young, "Foundations for the Arcadia environment architecture," in *Proceedings SIGSOFT '88: Third Symposium on Software Development Environments*, pp. 1–13, December 1988.

[25] L. A. Clarke, J. C. Wileden, and A. L. Wolf, "GRAPHITE; A meta-tool for Ada environment development," in *Proceedings of 2nd International Conference on Ada Applications and Environments*, pp. 81–90, April 1986.

[26] J. C. Corbett, "A tool for automatic elimination of constraints in constrained expression analysis," Constrained Expression Memorandum 90-2, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

[27] 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.

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

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

[30] G. S. Avrunin, "A prototype inequality generator," Software Development Laboratory Memo 88-1, Department of Computer and Information Science, University of Massachusetts, 1988.

[31] R. J. Dakin, "A tree search algorithm for mixed integer programming problems," *Computer Journal*, vol. 8, pp. 250–255, 1965.

[32] M. A. Saunders, "MINOS system manual," Tech. Rep. SOL 77-31, Stanford University, Department of Operations Research, 1977.

[33] J. Burnett and U. Buy, "Solving integer programming problems using the IMNOS prototype." In preparation.

[34] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. New York, New York: John Wiley & Sons, Inc., 1988.

[35] J. C. Corbett and G. A. Polk, "A tool for generating behaviors in constrained expression analysis," Constrained Expression Memorandum 90-3, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

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

[37] S. Tu, S. M. Shatz, and T. Murata, "Theory and application of Petri net reduction for Ada-tasking deadlock analysis." Submitted for publication, 1990.

[38] K.-C. Tai, "A graphical notation for describing executions of concurrent Ada programs," *Ada Letters*, vol. 6, pp. 94–103, January–February 1986.

[39] G. Ricart and A. K. Agrawala, "An optimal algorithm for mutual exclusion in computer networks," *Communications ACM*, vol. 24, pp. 9–17, 1981.

[40] L. Lamport, "A new solution of Dijkstra's concurrent programming problem," *Communications ACM*, vol. 17, no. 8, pp. 453–455, 1974.

[41] G. S. Avrunin and J. C. Wileden, "Describing and analyzing distributed software system designs," *ACM Trans. Programming Languages and Systems*, vol. 7, pp. 380–403, July 1985.

[42] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated constrained expression analysis of real-time software." Submitted, Oct. 1990.

[43] C. E. McDowell, "Representing reachable states of a parallel program," Computer Research Laboratory Technical Report USSC-CRL-89-17, University of California, Santa Cruz, August 1989.

[44] A. Valmari, "A stubborn attack on state explosion." To appear in *Proceedings of the DIMACS Workshop on Computer-Aided Verification*, New Brunswick, NJ, June 1990.

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

[46] E. M. L. Beale and J. A. Tomlin, "Special facilities in a general mathematical programming system for non-convex problems using ordered sets of variables," in *Proceedings of the Fifth International Conference on Operations Research* (J. Lawrence, ed.), 1969.

[47] A. I. Ali, J. Kennington, and B. Shetty, "The equal flow problem," *European J. Oper. Res.*, vol. 36, pp. 107–115, 1988.