# Automated Analysis of Concurrent Systems With the Constrained Expression Toolset

George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, *Member, IEEE*, and Jack C. Wileden, *Member, IEEE*

*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 completely automated analyses of a variety of concurrent systems, 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 involve hundreds of concurrent processes.**

*Index Terms*— **Concurrent systems, automated analysis, analysis tools, experimental evaluation, toolset performance, constrained expressions, formal methods, event-based model.**

## I. 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 the 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 constructing the set of possible states of the concurrent system (e.g., [1]–[3]), on proving theorems in some logical structure associated with the system (e.g., [4], [5]), and on examining the execution of a completed system or some simulation of it (e.g., [6], [7]).

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

Manuscript received December 10, 1990; revised June 10, 1991. Recommended by T. Murata. This work was partially supported by the ONR through Grant N00014-89-J-1064, and by the NSF through Grants CCR-8806970, CCR-8702905, and CCR-8704478 (with cooperation from DARPA (ARPA Order 6104)).

G. S. Avrunin, J. C. Corbett, and J. C. Wileden are with the University of Massachusetts, Amherst, MA 01003.

U. A. Buy is with the University of Illinois, Chicago, IL.

L. K. Dillon is with the University of California, Santa Barbara, CA 93106.

IEEE Log Number 9103527.

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 [8]–[11]. 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 the 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 [10], 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. Section III describes the tools and analysis methods they implement. Section IV discusses the results of our experiments with the application of the toolset. In Section V we assess the strengths and weaknesses of the toolset and our approach, on both theoretical and empirical grounds. In the last section we discuss our conclusions and some of our future research plans.

## II. 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. A detailed and rigorous presentation of the formalism is given in [10, appendix], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [9]. 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 which 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 [10]. 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 III-A.

The constrained expression formalism assumes an event-based model of computation. An execution of a concurrent system is modeled by a (totally or partially) 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.

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. 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*, 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 [9], [10].

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). 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* [12]. Informally, an interleaving set represents a partial order on

```
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;
```

Fig. 1. Fork and philosopher tasks from dining philosophers in CEDL.

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 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 representation of an execution as an interleaving set makes it possible to express the partial order which 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. The analysis techniques described below are fully compatible with an interpretation of executions as partial orders on events.

### A. 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 activities 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 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. Fig. 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. Fig. 2 gives the task expressions produced by the toolset for these two tasks, and Fig. 3 shows some of the constraints produced by the toolset for this system. We give the 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. (The task expressions and constraints required for CEDL are all regular, so the dagger operator does not appear.) For the example, we assume that the set of symbols appearing in an expression determines its alphabet. The table in Fig. 4 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. The synchronization constraint in Fig. 3 enforces proper synchronization of rendezvous for one of the entries of a fork task. Similar synchronization constraints are required for all entries. The blocking constraint in Fig. 3 ensures that a fork task does not wait forever for a rendezvous with one of the philosophers if the philosopher

```
(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)" ))))
```

Fig. 2. Two task expressions derived from the dining philosophers problem.

task is also waiting for the same rendezvous. The queueing constraint in that figure ensures that 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 III (see Fig. 7).

Under a partial order interpretation for the semantics of constrained expressions, 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 `P0` and `P1` each think and eat once and `P2` never does anything. 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 such partially ordered executions in which `P0` and `P1` each eat 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. That characterization of a system trace is consistent with the characterization above, provided that the alphabets of the task expressions are disjoint. 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.

### *B. 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

```
(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)" ))))

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

 (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)"))))
```

Fig. 3. Some constraints generated by the toolset from the dining philosophers system.

| *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) |

Fig. 4. Interpretation of event symbols.

dining philosophers, for example, the question of whether a philosopher who 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.

From the task expressions and constraints, we generate a system of inequalities involving the numbers of occurrences of the various symbols in a system trace. Additional inequalities can then be added to express the assumption that a specified symbol or pattern of symbols also occurs in a trace. If the resulting 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 it is automated in the constrained expression toolset.

## III. The Tools

There are five major components of the constrained expression toolset (see Fig. 5). In normal use, an analyst would first use the *deriver* to produce a constrained expression represen-

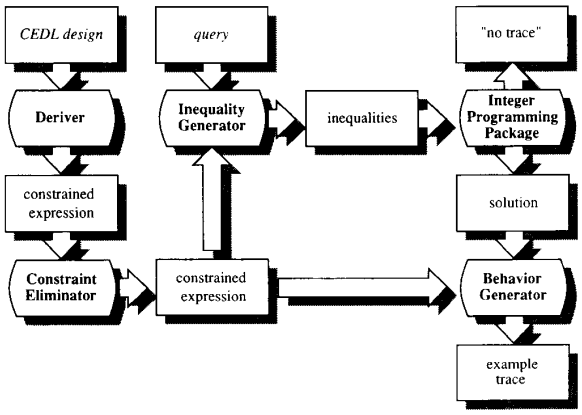



Fig. 5. Diagram of constrained expression toolset.

tation 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. The constraint eliminator, inequality generator, and behavior generator are written in Common LISP. The deriver is written in Ada and the integer programming package is written in FORTRAN.

In the remainder of this section we discuss each of the components of the toolset in more detail. Technical reports describing the implementation of the tools are available from the authors.

### A. The Deriver

The deriver 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.

Our current deriver requires that designs be written in CEDL, our Ada-like 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 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 [13]. Other restrictions limit the complexity of CEDL designs and 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 II-A. 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, using an attribute grammar approach. Fig. 2 shows two task expressions produced by the deriver from the CEDL code of Fig. 1. The deriver produces the constraints for the constrained expression representation of a CEDL design by instantiating a fixed set of constraint templates. Fig. 3 gives examples of the constraints produced by the deriver.

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 fact, as discussed below, the inequality generator and behavior generator rely on certain features of CEDL in order to improve efficiency.

### B. 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 which 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 which 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 to modify the constrained

```
("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)"))
```

Fig. 6. Part of the task expression for task T.

```
(defconstraint DATAFLOW_1
  ("STAR" ("OR" ("SEQUENCE" "def(flag;true)" ("STAR" "use(flag;true)"))
                ("SEQUENCE" "def(flag;false)" ("STAR" "use(flag;false)")))))
```

Fig. 7. Dataflow constraint for local variable flag.

```
("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)"))
```

Fig. 8. Part of task expression after elimination of dataflow constraint.

expression representations in such a way that the inequalities generated from them exclude such solutions.

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;
```

Fig. 6 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 Fig. 7 filters out these erroneous strings. The constraint allows any number of def(flag;*val*) symbols, each of which represents the assignment of the value *val* to the variable flag. It also allows each def(flag;*val*) symbol to be followed by any number of use(flag;*val*) symbols with that particular value, each representing a use of the variable, before the next def(flag;*val*) symbol. Any string satisfying both the task expression 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 satisfies both the old task expression and the constraints). Fig. 8 shows the result of incorporating the dataflow constraint for the variable flag into the task expression for task T shown in Fig. 6. 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 intratask constraints may be output either as regular expressions (RE's), deterministic finite automata (DFA's), or in a hybrid form we call regular-expression deterministic-finite automata (REDFA's). REDFA's are DFA's whose arcs are labeled with regular expressions satisfying certain conditions that preserve determinacy. We have found that it is easier to generate "efficient" inequality systems from RE's, but that, after constraint elimination, the RE's for some tasks are very much larger than their corresponding DFA's. The efficiency of an inequality system is, roughly speaking, the size of the task representation (RE, DFA, or REDFA) divided by the size of the inequality system (variables $\times$ inequalities). Unlike RE's, REDFA's are never significantly larger than the DFA's from which they are generated. Unlike DFA's, REDFA's allow easy generation of very efficient inequality systems.

To incorporate a set of intratask constraints into a task expression, all the regular expressions involved are converted to DFA's, which are then intersected pairwise. 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 that the constraint alphabet is a subset of the task alphabet, the result of the intersection is a DFA which accepts exactly those strings accepted by the original task DFA in which the symbols contained in the intratask 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 intratask constraints are exactly the dataflow constraints, since there are no global variables and all other constraints involve more than one task expression.

Using the intersection procedure described 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 intratask 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. It is exactly this state explosion we seek to avoid by considering the tasks separately and ignoring some of the dependencies among them.

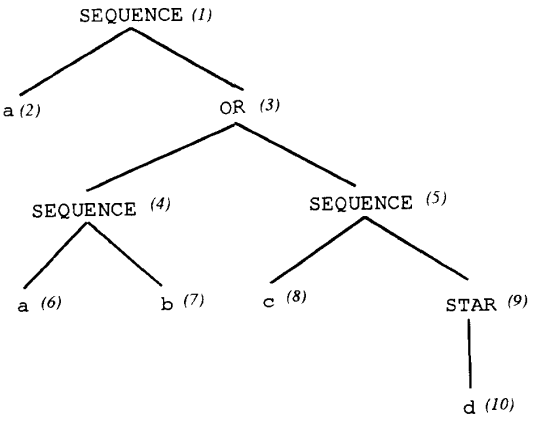



Fig. 9. Parse tree for the regular expression $a(ab \vee cd^*)$.

### C. 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.

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 DFA's or REDFA's. 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 II, 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 DFA's and REDFA's, and then discuss the 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 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 non-negative, 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 Fig. 9, 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, 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

bounded above by $B$, 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 IV.)

In this fashion we generate a system of linear inequalities from the task expressions. Our first prototype of the inequality generator used exactly this approach. The current inequality generator makes use of several optimizations which 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 REDFA's, some arcs are labeled by regular expressions rather than single-event symbols. For each such arc, we generate additional equations corresponding to the regular expression labeling 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. Fig. 10 shows a DFA accepting the language of the regular expression of Fig. 9. 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:

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

Note that the variable $x_5$ is unconstrained, since it appears only in the last equation and cancels out there. This is due to the fact that the corresponding arc, being a loop, is both incoming and outgoing. Essentially, the same phenomenon occurs with any cycle in the DFA and can lead to spurious solutions to the system of inequalities. This problem is related to the difficulty with Kleene stars noted above, and can be eliminated by introducing quadratic inequalities that ensure that no variable corresponding to an arc in the cycle can be nonzero unless the variable corresponding to some arc connecting a state outside the cycle to one in the cycle has a nonzero value. (As in the regular expression case, when all variables are bounded above, the effect of such quadratic inequalities can be achieved with linear ones.)

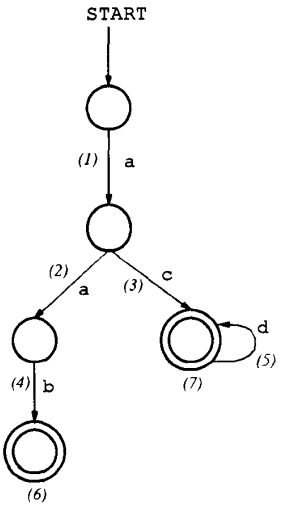



Fig. 10. DFA accepting the language of $a(ab \vee cd^*)$.

Having produced equations for each task, the inequality generator then begins 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 Fig. 9, 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 arcs.

To see how the constraints justify additional inequalities, consider first the synchronization constraint shown in Fig. 3. 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. The inequality generator therefore produces 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 blocking constraint of Fig. 3 and the fact that task expressions produced by the deriver have the property that each task contributes at most 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 generate inequalities involving the variables from the tasks directly from the CEDL constraint templates.

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.

### D. 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 [14]. Our implementation of this algorithm makes use of the MINOS [15] 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). 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.

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 which 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 IV, some problems have arisen with large systems of inequalities. We are therefore investigating approaches to integer programming which take advantage of the fact that our systems of inequalities can be regarded as network problems with side constraints.

### E. 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. Given the solution and the constrained expression (a set of task RE's, DFA's, or REDFA's 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, rather than from a regular expression.

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 DFA's for all the tasks and constraints (the behavior generator uses the DFA representation for all tasks and constraints, converting regular expressions to DFA's 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

| name | tasks | deriv | elim | ineq | IMINOS | behav | size | total |
|---|---|---|---|---|---|---|---|---|
| dp20 | 40 | 109 | 4 | 26 | 9 | 21 | 381 × 320 | 169 |
| dp40 | 80 | 203 | 11 | 77 | 71 | 46 | 761 × 640 | 408 |
| dp60 | 120 | 298 | 21 | 158 | 74 | 78 | 1141 × 960 | 629 |
| dp80 | 160 | 403 | 35 | 248 | 75 | 122 | 1521 × 1280 | 883 |
| dp100 | 200 | 501 | 60 | 399 | 120 | 169 | 1901 × 1600 | 1249 |
| dp20-h | 41 | 140 | 105 | 157 | 65 | | 603 × 1261 | 467 |
| dp30-h | 61 | 190 | 437 | 538 | 58 | | 903 × 2491 | 1223 |
| dp40-h | 81 | 265 | 1079 | 1516 | 81 | | 1203 × 4121 | 2941 |
| dp20-eh | 41 | 141 | 128 | 171 | 222 | 54 | 607 × 1305 | 716 |
| dp30-eh | 61 | 196 | 392 | 537 | 296 | 119 | 905 × 2523 | 1540 |
| dp40-eh | 81 | 259 | 1104 | 1603 | 865 | 239 | 1205 × 4163 | 4070 |
| dp5-u | 11 | 59 | 2 | 10 | 3 | 10 | 144 × 174 | 84 |
| dp5-efm | 6 | 49 | 83 | 57 | 6 | 18 | 303 × 585 | 213 |
| dp5-fm | 6 | 51 | 29 | 6 | 1 | | 73 × 95 | 87 |

Fig. 11. Toolset performance on several variations of dining philosophers problem.

in any successor of that global state, pruning the search tree. The search starts at the global state in which all task and constraint DFA's 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 until a *final global state* is found in which all task and constraint DFA's are in accepting states and all counts are zero or until all paths to a user-specified depth bound have been explored. 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 those 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 which allow the analyst to 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 occurring 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.

## IV. Experimental Results

As noted above, we believe that an 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 have tried to discuss the examples and our results in enough detail to show the effect of various factors on the performance of the constrained expression tools, although 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.

All the experiments reported in this paper were run on a DEC station 3100 with 24 megabytes of memory; times given are in CPU seconds on that machine and include both user and system time. The CEDL code for the examples discussed here is too long to include in this paper, but is available from the authors.

### A. Dining Philosophers

Perhaps the most widely known example in the concurrent systems literature is Dijkstra's dining philosophers. The system is interesting because of the possibility of deadlock. Various approaches can be used to prevent the deadlock.

Fig. 11 shows the performance of the constrained expression toolset on several variations of this system. In all cases, analysis is intended to detect the possibility of deadlock. The columns give, respectively, the name of the system, 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.

The first five rows of the figure give statistics for several sizes of the basic dining philosophers system. We model each fork and each philosopher by separate tasks, as illustrated in Section II-A. A system dp$n$ with $n$ philosophers thus has $2n$ tasks. For all of these systems, the toolset automatically produces a trace exhibiting deadlock.

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 modeled this in the systems dp20-h,

dp30-h, and dp40-h by introducing an additional host task and modifying the philosopher tasks. Control flow in the system with host depends on the value of a variable maintained by the host task which counts the number of philosophers in the 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. This process, however, together with the additional entry calls in the philosopher tasks, leads to significantly bigger systems of inequalities. Rows six through eight of the figure summarize the results of analyzing these systems with the toolset. In each case, IMINOS reports that there is no integral solution to the system of inequalities, implying that no deadlock is possible. It is therefore not necessary to run the behavior generator in these cases.

For comparison, we also analyzed systems of the same sizes in which the host erroneously allows all the philosophers to enter the dining room at once. The performance of the toolset on these problems is shown in the rows for the systems dp20-eh, dp30-eh, and dp40-eh. 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. For comparison with their published reports of automated analyses, we report briefly on the analysis of three of these with the constrained expression toolset.

The first of these systems, dp5-u, is a five-philosopher "unrolled" version of the dining philosophers with host, like that analyzed by Young *et al.* [16] using their CATS system. 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 system for deadlock. The design published in [16] 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 S. Shatz), and the constrained expression toolset produces a trace displaying the deadlock in the "unrolled" system.

The final two rows of the figure give results for dining philosophers systems similar to ones analyzed by Karam and Buhr [1] for deadlock and starvation. These systems use a single fork manager task to model the forks, rather than individual tasks. Deadlock is possible in the system dp5-efm, and the toolset produces a system trace that displays deadlock. The fork manager prevents deadlock in the system dp5-fm by requiring the philosophers to pick up both forks at the same time. In this case, IMINOS reports that no deadlock is possible, and it is not necessary to run the behavior generator.

All the IMINOS runs in Fig. 11 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 dp30 (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 V.

### B. Gas Station

The automated gas station example introduced by Helmbold and Luckham [6] has been studied by a number of authors (e.g., [1], [17]). This system models an automated gas station with an operator, a number of pumps, and a collection of customers. We have analyzed several versions of the system which correspond to some of the refinements used by Helmbold and Luckham. The performance of the toolset on these examples is reported in Fig. 12. The columns of the figure have the same significance as in Fig. 11.

In the first of our systems, gas2-e, there are two customer tasks, one pump task, and one operator task. In this version, a race condition can lead to deadlock, and our analysis detects this. Our second version, gas2, eliminates the race condition and the toolset correctly reports that the system cannot deadlock. (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 the deadlock-free two-customer design is scaled up to three customers, however, a more complicated race condition arises, again leading to the possibility of deadlock. (This was first noticed by K. C. Tai [18], who used a graphical analysis method to detect the error.) We analyzed two versions of the three-customer extension of this problem. The first, gas3, is a straightforward extension. In this case, the constraint eliminator produces an REDFA for the operator task that has a very large number of states due to the possible states of a queue of waiting customers. The large number of states (5239 in the DFA produced by the eliminator, 433 in the corresponding REDFA) is responsible for the fact that the eliminator takes more than 30 min 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 the first three-customer extension are shown in the third line of Fig. 12, and those for the version that reduces the number of states, gas3-res, are given in the fourth line. We note that these systems have many fewer tasks than the dining philosophers examples, but the systems of inequalities and the tool execution times are relatively large. This chiefly reflects the more complicated dataflow.

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 such systems, the number of states in the REDFA for the task representing the operator is much smaller than that in the versions discussed earlier. Results for these examples

| name | tasks | deriv | elim | ineq | IMINOS | behav | size | total |
|---|---|---|---|---|---|---|---|---|
| gas2-e | 4 | 36 | 26 | 11 | 5 | 8 | 120 × 200 | 86 |
| gas2 | 4 | 37 | 30 | 12 | 5 | | 125 × 209 | 84 |
| gas3 | 5 | 44 | 2034 | 202 | 361 | 195 | 604 × 1401 | 2836 |
| gas3-*res* | 5 | 46 | 644 | 63 | 164 | 53 | 315 × 643 | 970 |
| | | | | | | | | |
| gas2-s | 4 | 34 | 2 | 3 | 1 | | 65 × 63 | 40 |
| gas3-s | 5 | 45 | 13 | 7 | 8 | | 107 × 131 | 73 |
| gas2-se | 4 | 37 | 3 | 5 | 2 | 4 | 76 × 80 | 51 |

Fig. 12. Toolset performance on the gas station.

| name | tasks | deriv | elim | ineq | IMINOS | behav | size | total |
|---|---|---|---|---|---|---|---|---|
| rw-d | 6 | 40 | 6 | 6 | 3 | 3 | 82 × 137 | 58 |
| rw | 6 | 40 | 5 | 2 | | | | 47 |
| rw-p | 6 | 41 | 7 | 7 | 4 | | 90 × 148 | 59 |

Fig. 13. Toolset performance on readers and writers problem.

also appear in Fig. 12. Our analysis was again 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, gas2-s and gas3-s. (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 two-customer version of this example containing an error (similar to that in the two-customer version discussed previously) that permits deadlock to occur. Results for this system, gas2-se, are given in the last line of the figure.

### C. 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. 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 and relinquish 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.

Simultaneous access by a reader and a writer would be represented in a system trace by an occurrence of a symbol representing a writer gaining access between symbols representing a reader gaining and relinquishing access, or by the occurrence of a symbol representing a reader gaining access between symbols representing a writer gaining and relinquishing access. 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. We therefore modified the controller task so that each time a reader or writer gains access to the resource, it checks to determine whether a reader and a writer both have access, 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 with four readers and one writer are shown in Fig. 13. The first line gives times for an incorrect system in which an error in the controller task allows a deadlock. The second line gives results for a correct system which 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 in the figure 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 refusing read requests while a writer is waiting to gain access to the resource. This system, which is correct, was analyzed to detect deadlock. The toolset correctly reports that deadlock is impossible.

### D. 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 [19]. In it, 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

| name | tasks | deriv | elim | ineq | IMINOS | behav | size | total |
|---|---|---|---|---|---|---|---|---|
| ra1 | 6 | 46 | 11 | 4 | 3 | | 129 × 186 | 70 |
| ra3-e | 7 | 87 | 30 | 8 | 71 | 35 | 216 × 247 | 238 |
| ra3 | 7 | 85 | 30 | 8 | 72 | 60 | 216 × 247 | 262 |

Fig. 14. Toolset performance on the distributed mutual exclusion examples.

and are similar to the numbers used in Lamport's "bakery algorithm" [20].

The constrained expression approach was applied in [21] 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, and the analysis was carried out by hand. We have used the toolset to examine a similar design written in CEDL. Fig. 14 summarizes the results of these experiments.

We began by considering a design for a single-node system, ra1, in which the details of the Ricart–Agrawala algorithm had not yet been elaborated. The analysis was intended to determine whether a request received at the node may be permanently deferred. The toolset showed that this cannot happen. This is essentially equivalent to the analysis performed by hand in [21], although the different communication primitives in CEDL and DYMOL make the details of the designs quite different.

We next considered two versions of a system with three elaborated nodes and an additional task simulating the resource. In this case, we wanted 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 nodes 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 nodes 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 systems, ra3-e, we introduced a race condition which would allow simultaneous access to the resource by two nodes. IMINOS found a solution to the system of inequalities, but due to the problem with cycles in the REDFA discussed in Section III-C, this solution is spurious. We then manually added the linear inequalities necessary to exclude solutions which incorrectly give nonzero values for arcs in those cycles, as described in that section, and ran IMINOS again. (We believe that automating this process should be straightforward and expect to add this feature to the toolset in the near future.) IMINOS found another solution, and the behavior generator reported that this solution was also spurious. Examination of the output of the behavior generator showed 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 thus detected a possible deadlock of the three-node system due to the error, rather than the deferral of requests. The time shown in the table for the performance of IMINOS on ra3-e is for the second run, which is somewhat longer than for the run without the additional inequalities.

In the second version of this three-node system, ra3, the race condition is eliminated so that the resource is used in a mutually exclusive fashion. In this case as well, the problem with cycles leads to a solution which does not correspond to a behavior, and we manually added inequalities as before. IMINOS found a solution to the new system of inequalities (as above, the time shown for IMINOS is for the second, longer run). Again, the behavior generator correctly reported 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 nodes behaves as if a request from the other node was received before it decided to request the resource itself. 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.

### *E. Counters and Systems with Many Identical Tasks*

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 initial 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 Fig. 15 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

| cus | r1 | r2 | tasks | deriv | ineq | IMINOS | size | total |
|---|---|---|---|---|---|---|---|---|
| 500 | 490 | 490 | 502 | 25 | 3 | 2 | 36 × 39 | 30 |
| 500 | 490 | 489 | 502 | 25 | 3 | 2 | 36 × 39 | 30 |
| 1000 | 990 | 990 | 1002 | 25 | 3 | 2 | 36 × 39 | 30 |
| 1000 | 990 | 989 | 1002 | 25 | 3 | 2 | 36 × 39 | 30 |

Fig. 15. Toolset performance with many identical tasks.

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 give the results for systems with 500 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 1000 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.

## V. 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.

### *A. 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 min. 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 which involve hundreds of concurrent processes. This is in marked contrast to the results reported for most other methods which 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.

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 which 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 which might take better advantage of the special characteristics of our systems of inequalities. We are currently investigating these possibilities.

The performance of the toolset is not easily predicted from known results on the computational complexity of the algorithms it implements, especially since problems like the detection of deadlock are *NP*-hard [22]. 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 DFA's 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. The complexity of inequality generation 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, 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.

Thus 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.

### *B. 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 IV 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 [23], 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. 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 intratask 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 [16] 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. 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, as discussed in Section III-C. 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 DFA's or REDFA's, 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 STAR's or cycles in an *ad hoc* manner and it should be possible to automate this process in a fairly straightforward fashion. 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 STAR's 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. 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 Fig. 12, 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. In fact, the process of detecting such variables and resetting them to some value would be relatively easy to automate using dataflow analysis techniques, although as yet we have not attempted to incorporate such automated resetting of variables into our toolset.

### C. 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 a reachability graph for states of the system (e.g., [1], [2], [16]). In general, the number of states such methods must examine is exponential in the number of tasks in the system, and different approaches are taken to reducing this complexity. For example, the CATS system [16] 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 [17] 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 50–100 tasks. Similarly, Young *et al.* 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 efficient. 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 [24], 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 variable to $n$ rather than one, as discussed in Section IV-E.) Valmari [25] has described a method which can detect deadlock in systems with a communication structure like that of the basic dining philosophers in time which 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 which can handle systems with more than 100 tasks.

Another approach, very closely related to ours, is the Petri-net invariant method of Murata *et al.* [26]. 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 [26] 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.

## VI. 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 which 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 IV, indicate that the constrained expression toolset can be used to analyze systems involving several hundred tasks. The toolset carries out a completely automated analysis, starting from the 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 cannot directly address 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. 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.

We are working on improvements or extensions to every component of the toolset. Many of these modifications are aimed at more fully automating the analysis of systems with large numbers of identical tasks described previously. Others are intended to improve the inequality solving component of the toolset, first by improving the heuristics used in our current version of IMINOS, and later by replacing IMINOS with a special-purpose integer linear programming system that can exploit the special structure found in the inequality systems that our tools generate. Still others will make the behavior generator more efficient and more helpful in cases where IMINOS finds spurious solutions.

Our work on extending the constrained expression formalism and analysis techniques will 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 have also recently developed and begun experimenting with an extension of the constrained expression analysis techniques which can be used to assess the timing properties of concurrent systems and have extended the toolset to implement this technique. Details of this approach to constrained expression analysis of real-time systems and an example of its application can be found in [23].

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 Eng.*, vol. 16, pp. 829–843, Aug. 1990.

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

[3] R. N. Taylor, "A general-purpose algorithm for analyzing concurrent programs," *Commun. 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 Syst.*, vol. 8, pp. 244–263, Apr. 1986.

[5] L. K. Dillon, "Verifying general safety properties of Ada tasking programs," *IEEE Trans. Software Eng.*, vol. 16, pp. 51–63, Jan. 1990.

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

[7] D. S. Rosenblum and D. C. Luckham, "Testing the correctness of tasking supervisors with TSL specifications," in *Proc. ACM SIGSOFT '89 3rd Symp. on Software Testing, Analysis and Verification*, R. A. Kemmerer, Ed., pp. 187–196 (also published in *Software Eng. Notes*, vol. 14, no. 8, 1989).

[8] J. C. Wileden, "Constrained expressions and the analysis of designs for dynamically-structured distributed systems," in *Proc. Int. Conf. on Parallel Process.*, Aug. 1982, pp. 340–344.

[9] 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 Eng.*, vol. SE-12, pp. 278–292, Feb. 1986.

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

[11] G. S. Avrunin, L. K. Dillon, and J. C. Wileden, "Experiments with automated constrained expression analysis of concurrent software systems," in *Proc. ACM SIGSOFT '89 3rd Symp. on Software Testing, Analysis and Verification*, R. A. Kemmerer, Ed., pp. 124–130 (also published in *Software Eng. Notes*, vol. 14, no. 8, 1989).

[12] S. Katz and D. Peled, "An interleaving set temporal logic," in *Proc. 6th Ann. ACM Symp. on Principles of Distributed Comput.*, 1987, pp. 178–190.

[13] L. A. Clarke, J. C. Wileden, and A. L. Wolf, "Nesting in Ada programs is for the birds," in *Proc. ACM-SIGPLAN Symp. on the Ada Program. Language*, 1980, pp. 139–145 (also published in *SIGPLAN Notices*, vol. 15, no. 11, 1980).

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

[15] M. A. Saunders, "MINOS system manual," Dept. Operations Res., Stanford Univ., Palo Alto, CA, Tech. Rep. SOL 77–31, 1977.

[16] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck, "Integrated concurrency analysis in a software development environment," in *Proc. ACM SIGSOFT '89 3rd Symp. on Software Testing, Analysis and Verification*, R. A. Kemmerer, Ed., pp. 200–209 (also published in *Software Eng. Notes*, vol. 14, no. 8).

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

[18] K.-C. Tai, "A graphical notation for describing executions of concurrent Ada programs," *Ada Lett.*, vol. 6, pp. 94–103, Jan.–Feb. 1986.

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

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

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

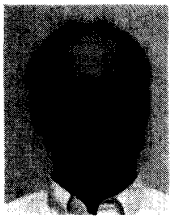
[22] R. N. Taylor, "Complexity of analyzing the synchronization structure of concurrent programs," *Acta Inform.*, vol. 19, pp. 57–84, 1983.

[23] G. S. Avrunin, J. C. Corbett, L. K. Dillon, and J. C. Wileden, "Automated constrained expression analysis of real-time software," Dept. Comput. and Inform. Sci., Univ. Massachusetts, Amherst, Tech. Rep. 90-117, Dec. 1990.

[24] C. E. McDowell, "A practical algorithm for static analysis of parallel programs," *J. Parallel and Distributed Process.*, vol. 6, pp. 515–536, June 1989.

[25] A. Valmari, "A stubborn attack on state explosion," in *Computer-Aided Verification '90* (Series in Discrete Mathematics and Theoretical Computer Sci., vol. 3), E. M. Clarke and R. P. Kurshan, Eds. Providence, RI: Amer. Math. Soc., 1991, pp. 25–41.

[26] T. Murata, B. Shenker, and S. M. Shatz, "Detection of Ada static deadlocks using Petri net invariants," *IEEE Trans. Software Eng.*, vol. 15, pp. 314–326, Mar. 1989.

**George S. Avrunin** received the B.S., M.A., and Ph.D. degrees in mathematics from the University of Michigan.

He is a Professor in the Department of Mathematics and Statistics at the University of Massachusetts at Amherst. In addition to formal methods and tools for the analysis of concurrent and real-time software systems, his research interests include the cohomology and representations of finite groups.

Dr. Avrunin is a member of the American Mathematical Society, the Association for Computing Machinery, the Association for Women in Mathematics, and the IEEE Computer Society.

**Ugo A. Buy** received the Laurea degree in electrical engineering from the Politecnico di Milano, Milan, Italy, in 1980, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst in 1983 and 1990, respectively.

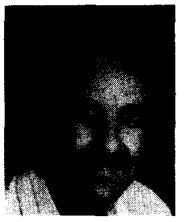
From 1983 to 1986 he was a Software Engineer with the Digital Equipment Corporation, Hudson, MA. He is currently an Assistant Professor with the University of Illinois at Chicago. His research interests focus on defining formal techniques and developing automated tools for the analysis of concurrent and real-time systems. His additional interests include specification methods and automatic code generation for concurrent programs.

**James C. Corbett** received the B.S. degree in computer science from Rensselaer Polytechnic Institute, Troy, NY, in 1987, and the M.S. degree in computer science from the University of Massachusetts at Amherst, where he is currently completing the Ph.D. degree requirements. His research interests are in the analysis and verification of concurrent and real-time systems.

**Laura K. Dillon** (S'81–M'84) received the B.A. and M.S. degrees in mathematics from the University of Michigan, Ann Arbor, and the M.S. and Ph.D. degrees in computer science from the University of Massachusetts at Amherst.

She is an Associate Professor in the Department of Computer Science at the University of California, Santa Barbara. Her research interests include formal methods for analysis of concurrent software systems, software specification and verification, and programming languages. Her research focuses on providing automated support for reasoning about the behavior of software systems.

Dr. Dillon is a member of the Association for Computing Machinery, the IEEE Computer Society, and Computer Professional for Social Responsibility.



**Jack C. Wileden** (S'77–M'78) received the A.B. degree in mathematics and the M.S. and Ph.D. degrees in computer and communication sciences from the University of Michigan, Ann Arbor.

He is a Professor in the Department of Computer and Information Science at the University of Massachusetts at Amherst, and is a Director of the Software Development Laboratory. His research interests center on integrated software development environments, especially object management capabilities for environments, and on development tools and techniques applicable to concurrent software systems.

Dr. Wileden is a member of the Association for Computing Machinery and the IEEE Computer Society. He is currently serving as an Associate Editor of the IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS and as an IEEE Distinguished Visitor.