Automated Analysis of Concurrent and Real-Time Software

George S. Avrunin*
Department of Mathematics and Statistics
University of Massachusetts
Amherst, MA 01003

Jack C. Wileden*,†
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

This paper surveys the current status of our work on automated analysis of the logical and timing properties of concurrent software based on the constrained expression approach. It describes our analysis toolset, reports some extremely encouraging results of using the toolset to analyze logical properties of nontrivial concurrent systems, and discusses the modifications we have made to the toolset to apply it to analyzing timing properties. It then outlines ongoing and planned research directed at further improving these methods.

INTRODUCTION

Software systems can only be made truly robust and reliable if sufficiently powerful analysis techniques are made available to software developers and maintainers. Ideally, such analysis techniques should be applicable throughout the development of a software system, from its initial specification through its

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

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

design and coding, and also during its subsequent lifetime, when they would greatly aid its maintenance and modification. Therefore, these analysis techniques should be applicable to a wide range of software system descriptions, not just a single specification, design or programming language. Obviously, to be of the greatest value, the techniques should also be applicable to the broadest possible range of program structures and organizations. Finally, the techniques must be able to analyze systems of realistic size in a reasonable amount of time. This almost certainly requires that the techniques be automatable and computationally tractable.

Traditionally, the primary analysis problem for software developers and maintainers has been to assess *logical properties* of a software system. These logical properties include such things as whether the system will compute the intended results and whether the system's computation will terminate. A variety of analysis techniques, ranging from program testing to program proving, have been proposed and used, with greater or lesser success, to assess logical properties of software. Most programs represent a very large number of distinct possible sequences of statement executions, and most of these analysis techniques depend upon reasoning about or examining as many of those distinct sequences as possible. Generally speaking, they have suffered from limited applicability, computational intractability, or both.

Difficult though it is to assess the logical properties of a single, sequential program running on a single processor, it is significantly more difficult to assess the logical properties of *concurrent* or *distributed* software systems. Such systems consist of several programs running simultaneously, either in a logical sense, by having their executions interleaved on a single processor, or by actually executing on several interconnected computers running in parallel. Moreover, the behavior of such systems is often nondeterministic. As a result, concurrent or distributed software systems typically represent even larger numbers of distinct possible sequences of statement executions than do sequential programs. Furthermore, concurrent programs have additional logical correctness properties, such as freedom from deadlock or mutually exclusive use of shared resources, that must be assessed. It is these attributes that make analyzing concurrent or distributed software even more difficult than analyzing sequential software. Not surprisingly, the analysis techniques that have been proposed for concurrent and distributed software have suffered heavily from limited applicability and computational intractability.

Several years ago, we set out to develop analysis techniques for concurrent

and distributed software systems that would be as broadly applicable as possible and that would be sufficiently computationally tractable that they could be used to analyze realistic problems concerning realistic software systems. The result of that effort is a collection of analysis techniques based on the *constrained expression* formalism. These techniques have recently been automated in the form of a prototype toolset, and we have begun to achieve some very encouraging results from applying the toolset to various standard concurrent system problems. Most notably, the toolset demonstrates impressive performance on nontrivial problems and the degradation in its performance as the size of the system being analyzed increases is much less than that of most other approaches. This suggests that analysis of realistic problems and systems using our techniques may be a real possibility.

Real-time software systems, which are those software systems whose correctness depends upon timing properties as well as logical properties, pose an additional set of challenges for software analysis techniques. A central requirement for correct operation of a real-time software system is that the software have predictable timing properties. Of course, the traditional concern for logical correctness (i.e., that the software will compute the intended results) applies to real-time software as well. However, the additional property characterizing correctness for real-time software is that the maximum time required to produce the intended results can be guaranteed to be within some specified bound. The requirement of predictable timing properties implies a need for analysis techniques that can accurately assess the timing, and timing-related (e.g., resource demands), properties of software systems. As with logical properties, analyzing timing properties of concurrent or distributed software is even more difficult than performing similar analysis of sequential software.

We have recently developed and begun experimenting with a technique for assessing timing properties of concurrent, real-time software. This technique is an outgrowth of our work on the constrained expression techniques for analyzing logical properties of concurrent software systems. The results of our initial efforts to apply our constrained expression analysis techniques to real-time system analysis problems have been very encouraging. By extending the constrained expression formalism slightly and modifying parts of the prototype toolset we are now able to carry out an automated derivation of an upper bound on the time that can elapse between the occurrence of any designated pair of events in a concurrent system's behavior. Since the constrained expression techniques were explicitly tailored to be applicable to a wide range of software

descriptions, program structures and organizations, techniques and tools resulting from extending them to real-time systems should have those same desirable attributes.

In this paper, we begin by briefly describing the constrained expression formalism and analysis techniques. We then sketch the prototype toolset that automates the analysis techniques and give some representative highlights from our experimentation with the use of the toolset in analyzing logical properties of concurrent systems. Next we explain the modifications and extensions required to apply the constrained expression techniques and toolset to the analysis of timing properties. Finally, we discuss ongoing research aimed at improving our abilities to analyze both logical and timing properties of concurrent systems.

CONSTRAINED EXPRESSIONS

In the constrained expression approach to analysis of concurrent systems, the system descriptions produced during software development (e.g., designs in some design notation) are translated into formal representations, called *constrained expression representations*, to which a variety of analysis methods are then applied. This approach allows developers to work in the design notations and implementation languages most appropriate to their tasks. Rigorous analysis is based on the constrained expression representations that are mechanically generated from the system descriptions created by software developers.

This section contains a brief overview of the constrained expression formalism. Detailed and rigorous presentations of the formalism appear in [12] and in the appendix to [14], while less formal treatments intended to provide a more intuitive understanding of the features of the formalism appear in [7] and [4]. The use of constrained expressions with a variety of development notations is illustrated in [7] and [14]. A detailed discussion of the relation between constrained expressions and a variety of other methods for describing and analyzing concurrent software systems can be found in [7] and [17].

The constrained expression formalism treats the behaviors of a concurrent system as sequences of events. These events can be of arbitrary complexity, depending on the system characteristics of interest and the level of system description under consideration. Associating an *event symbol* to each event, we can regard each possible behavior of the system as a string over the alphabet of event symbols. While this suggests that behaviors must be viewed as total orders on events, in fact the constrained expression formalism is consistent with

viewing concurrent system behaviors either as total orders or as partial orders on events, as discussed in [4].

We use interleaving to represent concurrency. Thus, a string representing a possible behavior of a system that consists of several concurrently executing components is obtained by interleaving strings representing the behaviors of the components. The events themselves are assumed to be atomic and indivisible. "Events" that are to be explicitly regarded as overlapping in time are represented by treating their initiation and termination as distinct atomic events.

The set of strings representing behaviors of a particular concurrent system is obtained by a two-step process. First, a regular expression, called the *system expression*, is derived from a description of the system in some notation such as a design or programming language. The language of this expression includes strings representing all possible behaviors of the system. It may, however, also include strings that do not represent possible behaviors, as the system expression does not encode the full semantics of the system description. This language is then "filtered" to remove such strings, using other expressions, called *constraints*, which are also derived from the original system description. A string survives this filtering process if its projections on the alphabets of the constraints lie in the languages of the constraints. The constraints (which need not be regular) enforce those aspects of the semantics of the design or programming language, such as the appropriate synchronization of rendezvous between different tasks or the consistent use of data, that are not captured in the system expression. The reasons for this two-step process, which might not seem as straightforward as generating behaviors directly from a single expression, are discussed in [14], while an equivalent and more uniform interpretation process for constrained expressions is presented in [4].

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 string representing a possible behavior of the system. For example, questions about whether the system can deadlock might be phrased in terms of the occurrence of symbols representing the permanent blocking of components (e.g., processes or tasks) of the system.

Starting from the assumption that the specified symbol, or pattern of symbols, does occur in such a string, we use the form of the system expression and the constraints to generate inequalities involving the numbers of occurrences of various event symbols in segments of the string. If the system of inequali-

ties thus generated is inconsistent, the original assumption is incorrect and the specified symbol or pattern of symbols does not occur in a string corresponding to a behavior of the system. If the inequalities are consistent, we use them in attempting to construct a string containing the specified pattern.

In summary, the constrained expression approach is applicable to systems expressed in a variety of notations and languages. It offers a focused approach to analysis, which, by keeping the amount of uninteresting information produced to a minimum, can be very efficient. Over the last several years, we have developed a set of tools automating various aspects of constrained expression analysis and have achieved very good results in using them to analyze logical properties of a range of concurrent system examples [4]. We briefly describe the current version of the prototype toolset before discussing the application of the formalism, analysis techniques and toolset to analyzing both concurrent and real-time systems.

THE CONSTRAINED EXPRESSION TOOLS

The prototype toolset (see Figure 1) consists of five major components: a *deriver* that produces constrained expression representations from concurrent system designs in a particular design language; a *constraint eliminator* that replaces a constrained expression with an equivalent one involving fewer constraints; an *inequality generator* that generates a system of inequalities from the constrained expression representation of a concurrent system; an *integer programming package* for determining whether this system of inequalities is consistent or inconsistent, and, if the system is consistent, for finding a solution with appropriate properties; and a *behavior generator* that uses the constrained expression produced by the constraint eliminator plus the solution found by the integer programming package (when the inequalities are consistent) to try to produce a string of event symbols corresponding to a system behavior with the desired properties. The organization of the toolset is illustrated in the figure. We give brief descriptions of the tools and their use below. A more detailed discussion of the toolset and its implementation appears in [4].

The current toolset is intended for use with designs written in the Ada-based design language CEDL (Constrained Expression Design Language) [13]. CEDL focuses on the expression of communication and synchronization among the tasks in a distributed system, and language features not related to concurrency are kept to a minimum. Thus, for example, data types are limited, but most of

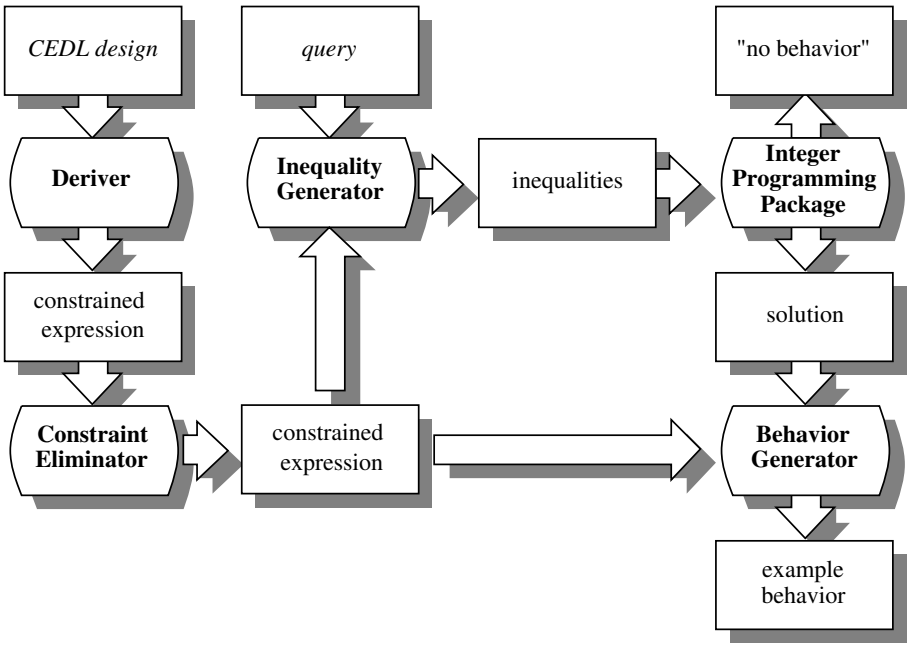

Figure 1: Diagram of Constrained Expression Toolset

the Ada control-flow constructs have correspondents in CEDL. We originally chose to work with a design notation based on Ada because Ada is one of the few programming languages in relatively widespread use that explicitly provides for concurrency, and because we expect our work on analysis of designs to contribute to and benefit from the Arcadia Consortium's work on Ada software development environments [16]. Despite Ada's shortcomings as a language for real-time software [1], we have used CEDL and the existing deriver in our initial experiments with analysis of timing properties.

The deriver [2] produces constrained expression representations from CEDL system designs. The system expressions it produces consist of the interleave of regular expressions, called *task expressions*, representing the behavior of the various components (called *tasks* in Ada, and hence in CEDL, terminology) of the concurrent system. The deriver also generates all required constraints.

The constraint eliminator [10] takes a subexpression of the system expres-

sion and certain constraints, and produces a new expression whose language is the set of strings in the language of the subexpression that satisfy the constraints. It requires that the subexpression and the constraints be regular and not involve the interleave operator. We typically use the constraint eliminator with a task expression and constraints that enforce correct dataflow within that task. The constraint eliminator converts the task expression and constraints into deterministic finite state automata (DFAs), and uses a procedure based on standard DFA intersection algorithms to produce a new automaton accepting only that subset of the language of the task expression that also satisfies the constraints. In earlier versions of the toolset, the new automaton was converted back into a regular expression, and the original task expression and constraints were eliminated from the constrained expression and replaced by this new task expression. Although very compact systems of inequalities can be generated from regular expressions, the conversion of the DFA resulting from constraint elimination into a regular expression sometimes results in enormous regular expressions. Our student, James Corbett, investigated this problem [9], and determined that, in certain cases, a more compact system of inequalities can be generated from the DFA representation of a task than from the regular expression into which that DFA would be converted. (In addition, this eliminates the cost of converting the DFA into a regular expression.) He also introduced a hybrid form we call REDFAs. We have found that, in general, it is best to work from the REDFA representation of tasks. The current implementation of the constraint eliminator, therefore, can replace the eliminated task expression with a regular expression, a DFA, or an REDFA, as specified by the user.

The inequality generator [3] takes a constrained expression representation as input. For each task, the inequality generator produces a collection of linear equations involving variables representing the number of times a node in a parse-tree of the task expression (or an arc in a DFA or REDFA representation of the task) is traversed in a behavior of the system. It then generates linear inequalities in these variables 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, for reasons of efficiency, the generation of inequalities from constraints depends on features of CEDL.

The constraints impose restrictions on the order and number of occurrences of event symbols in behaviors of the system. The integer programming variables we use represent only the total numbers of occurrences of symbols (or,

more precisely, of traversals of nodes in the parse-trees or arcs in finite state automata) and do not reflect the order in which those symbols occur. The inequalities we generate therefore do not directly reflect information about the order in which the various symbols occur. Similarly, the fact that we restrict ourselves to linear systems of equations and inequalities (in order to avoid the much more difficult computational problems of nonlinear systems) means that our systems do not fully reflect the semantics of the Kleene star operator (or, equivalently, of cycles in task DFAs). For these reasons, which are discussed more fully in [4], our systems of inequalities should be regarded as expressing necessary conditions that must be satisfied by any behavior of the concurrent system.

The inequality generator also provides an interactive facility allowing the analyst to add additional inequalities representing assumptions or queries about the behavior of the system and a reporting facility for use by a human analyst interpreting output of the integer programming package.

The integer programming package, called IMINOS [8], is a branch-and-bound integer linear programming system that we have implemented on top of the MINOS optimization package [15]. When the generated system of inequalities is consistent, the integer programming package produces a solution giving counts for the number of occurrences of the various event symbols. The behavior generator [11] uses heuristic search techniques to find a string of event symbols having the given counts and corresponding to a system behavior, helping the analyst to understand the solution found by the integer programming package. The behavior generator may also be used by the analyst for interactive exploration of the system.

ANALYSIS OF LOGICAL PROPERTIES

We have used the prototype constrained expression toolset to analyze a number of standard problems from the concurrent systems literature. We report here the results of analysis of several versions of the dining philosophers problem in order to give some idea of the capabilities and performance of the toolset. Additional results on these problems and others, together with a more detailed and complete discussion, appear in [4].

Perhaps the most widely known example in the concurrent systems literature is Dijkstra's dining philosophers problem, in which a group of philosophers sit at a round table with one seat for each philosopher and one fork be-

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
60	120	298	21	158	74	78	1141×960	629
80	160	403	35	248	75	122	1521×1280	883
100	200	501	60	399	120	169	1901×1600	1249
20	41	140	105	157	65		603×1261	467
30	61	190	437	538	58		903×2491	1223
40	81	265	1079	1516	81		1203×4121	2941
20	41	141	128	171	222	54	607×1305	716
30	61	196	392	537	296	119	905×2523	1540
40	81	259	1104	1603	865	239	1205×4163	4070

Figure 2: Toolset Performance on Versions of the Dining Philosophers Problem

tween each pair of philosophers. The philosophers alternately think and eat. A philosopher requires two forks to eat, and each philosopher who wants to eat attempts to pick up one fork, say the one on the left, and then the other. Having acquired both forks, the philosopher eats and then puts the forks down. The system is interesting because of the possibility of deadlock caused by all the philosophers picking up the forks on their left, leaving each of them unable to pick up a second fork. Various approaches can be used to prevent the deadlock.

We have analyzed several variations of this system. In the basic one, we model each fork by a task with two entries. Calls to the "up" entry represent the fork being picked up by a philosopher and calls to the "down" entry represent the fork being put down. Each fork task loops forever, accepting calls first at its up entry and then at its down entry. Each philosopher is represented by a task that repeatedly calls the up entry of the fork to its "left", the up entry of the fork to its "right", and then the down entries of the two forks. A system with n philosophers thus has $2n$ tasks. Our analysis is intended to detect the possibility of 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 by introducing an additional host task and modifying the philosopher tasks. The host task has two entries, "enter" and "leave", and a philosopher must rendezvous with the host at "enter" before attempting to pick up the first fork. After putting down the second fork, the philosopher calls the "leave" entry. The host keeps track of

the number of philosophers in the dining room (the number of rendezvous that have occurred at "enter" minus the number at "leave") and repeatedly accepts calls at "enter" as long as no more than $n - 2$ philosophers are in the dining room. The "leave" entry is unguarded, so calls at that entry can be accepted at any time.

Although the dining philosophers system with host and n philosophers involves only one more task than the basic system with the same number of philosophers, control flow in the host task depends on the value of the variable counting the number of philosophers in the dining room. The constraint eliminator intersects the task expression for the host with the constraint involving this variable, so that the system of inequalities properly reflects the dependence of control flow on the number of philosophers in the dining room and the analysis does not spuriously report deadlock. This process, however, together with the additional entry calls in the philosopher tasks, results in significantly bigger systems of inequalities.

In Figure 2, we show the performance of the constrained expression toolset in analyzing these dining philosophers problems. The columns in the table give, respectively, the number of philosophers, the number of tasks in the system, the time in seconds used by the deriver, the eliminator, the inequality generator, IMINOS, and the behavior generator, the size of the system of inequalities (number of inequalities $\times$ number of variables), and the total time used by the toolset. All the experiments reported in this paper were run on a DECstation 3100 with 24 MB of memory; times given are in CPU seconds on that machine and include both user and system time. The first three lines of the table give results for versions of the basic dining philosophers system with 60, 80, and 100 philosophers, respectively. In these examples, the toolset produces a behavior displaying the deadlock. The next three lines of the table give results for versions of the dining philosophers problem with a host task and 20, 30, and 40 philosophers respectively. In these examples, IMINOS correctly reports that deadlock is impossible and it is not necessary to run the behavior generator. The last three lines of the table give times for 20-, 30-, and 40-philosopher examples with host in which an erroneous bound in the host task allows all the philosophers into the room at the same time. In these cases, the toolset produces a behavior displaying the deadlock.

As the results in Figure 2 illustrate, the constrained expression toolset is capable of analyzing large systems. The toolset carries out a complete analysis of the basic dining philosophers problem with 100 philosopher tasks and 100 fork

tasks, starting from the CEDL code and producing a behavior displaying deadlock, in less than 21 minutes. When the behavior of the individual tasks is more complex, the toolset cannot handle quite so many tasks, but it is clear that it can be used with at least some systems that approach, or even exceed, realistic sizes for concurrent system designs. By way of comparison, we know of few other automated analysis techniques that are capable of handling versions of the basic dining philosophers problem with as many as 10 philosophers, and their execution time typically increases exponentially with the number of philosophers.

With some other examples, however, the integer programming component of the toolset finds solutions that do not correspond to behaviors of the concurrent system being analyzed. This is due to the fact that our systems of inequalities do not fully reflect the semantics of the constrained expression representation of that concurrent system, giving instead only necessary conditions for a sequence of events to correspond to a system behavior. In these cases, the behavior generator reports that the solution does not correspond to a behavior, but the analysis is inconclusive because there may be other solutions to the system of inequalities that do correspond to behaviors. In some cases, it is possible to deal with these problems in an *ad hoc* manner.

The current implementation of the toolset is not able to address questions involving fairness and can address certain questions about the order of event occurrences only indirectly by transforming the question into one involving the number of occurrences of other events. These problems are the subject of ongoing research.

ANALYSIS OF TIMING PROPERTIES

The constrained expression formalism and the toolset described in the preceding sections were originally developed to analyze logical properties of behaviors of concurrent or distributed systems. As a result, the formalism models computation as a stream of non-overlapping atomic events, with no notion of time, and the toolset is oriented toward finding (or disproving the existence of) complete behaviors that have some specified property, such as deadlock.

To apply the constrained expression analysis techniques to real-time systems, the formalism must first be extended to account for time. This can be done straightforwardly by assigning a duration to each event. The time required for a sequence of events is then just the sum of the durations of the individual events in the sequence. However, such an interpretation only makes sense when

the events are non-overlapping, as would be the case if the concurrent system being analyzed were to be run on a single processor. We adopt this straightforward extension to the formalism, and the corresponding limitation on the class of concurrent systems whose timing properties we can analyze, as a first step toward applying our constrained expression techniques to the analysis of real-time systems. Ongoing research is directed toward applying our techniques to "truly concurrent" (i.e., multiprocessor) systems.

We are interested in answering questions of the form "What is the longest time that can elapse between an occurrence of event A and the next occurrence of event B in a behavior of the system?" Such questions involve subsequences of events that might occur within the full sequences that correspond to complete system behaviors. (Of course, sometimes the subsequence of interest is the full sequence.) Our initial approach to applying the constrained expression toolset to the analysis of timing properties [6] was to manually modify the constrained expression generated by the deriver and constraint eliminator so that it represented (approximately) the subsequences of interest. In effect, this transformed questions about partial behaviors of the system into questions about complete behaviors of the system represented by the modified constrained expression. We then used the inequality generator and the integer programming package to find a bound on the duration of these behaviors.

The investigation of the DFA and REDFA forms for representing task expressions, however, suggested an approach that has allowed us to generate inequalities describing subsequences of behaviors directly from the original constrained expression, and we have extended the toolset to implement it. We now give an outline of this new approach. Details of our method and an example of its application can be found in [5]. We note that, although these extensions to the toolset were motivated by our interest in analyzing timing properties, they will also be valuable for analyses of logical properties that involve consideration of partial behaviors, such as detecting violations of mutual exclusion.

To explain our method for generating inequalities describing subsequences of behaviors, it is helpful to begin by discussing how we generate inequalities describing complete behaviors from DFA representations of tasks. Given a DFA accepting the language of a task expression, the basic approach is to assign a variable to each arc of the DFA. The value of the variable associated with an arc gives the number of times that arc is traversed in a particular behavior of the system. We then generate a "flow equation" for each state other than the initial and accepting states. This equation says that the sum of the variables as-

sociated with arcs into that state must equal the sum of the variables associated with arcs leaving the state. The initial state and the accepting states must be treated specially, of course. For the initial state, we set the sum of the variables associated with arcs leaving the state to 1, representing fact that the task is activated once and begins its computation in the initial state of the DFA. The flow equations then imply that the sum of the variables associated with arcs into the accepting states is 1, corresponding to the fact that the task ends its computation in exactly one accepting state.

In a subsequence of an event sequence corresponding to a complete behavior of a concurrent system, of course, the state of a particular task DFA at the beginning of the subsequence need not be its initial state and its state at the end of the subsequence need not be an accepting state. So we need to add an extra "flow in" at any state in which the DFA could be at the start of the subsequence, and an extra "flow out" at any state in which the DFA could be at the end of the subsequence. We do this by introducing additional variables representing the state of the DFA at the start and end of the subsequence, and generating additional equations reflecting the fact that the DFA is in exactly one state at each of those times.

The idea is as follows. Assume that each of the tasks of the concurrent system is represented by a task DFA, and that we are interested in finding an upper bound on the time between the occurrence of an event A in one task and the next occurrence of event B, possibly in a different task. For simplicity, we will restrict our attention to subsequences in which A does not occur again before B; we are thus asking for an upper bound on the duration of the subsequences of behaviors beginning with A and ending with B, and having no other occurrences of A or B. We will also assume that the events A and B each occur in exactly one task. This can easily be arranged by using unique names for event symbols corresponding to the events in different tasks.

We generate equations for each task DFA. As in the analysis of complete behaviors, we assign a variable x_a to each arc a in the task DFAs. We then assign a *start variable* s_i to each state i. This variable will be 1 if the task is in state i at the beginning of the subsequence, and 0 otherwise. If the symbol corresponding to A appears as a label on an arc in the DFA, we omit the start variables on all states except those with an outgoing arc labeled by the symbol corresponding to A, since we want the first event in the subsequence to be A. Similarly, we assign a *halt variable* h_i to each state i, but in the DFA in which the symbol corresponding to B occurs, we omit the halt variables for states not

having an incoming arc with this symbol. The variable h_i will be 1 if the task is in state i at the end of the subsequence, and 0 otherwise.

We then write flow equations for each state, counting s_i as flow into state i and h_i as flow out. These equations have the form

$$s_i + \sum_{a \in \text{In}(i)} x_a = h_i + \sum_{a' \in \text{Out}(i)} x_{a'},$$

where $\text{In}(i)$ is the set of arcs into state i and $\text{Out}(i)$ is the set of arcs out of state i. We then write equations stating that the sum of the start variables in each task DFA is equal to one. As before, the flow equations then imply that the sum of the halt variables in each task DFA is one.

Finally, we generate an additional equation for each pair consisting of an entry in the concurrent system and a task calling that entry. This equation expresses the fact that the number of times the given task calls that entry must be equal to the number of times a rendezvous with that task is accepted at the entry. This equation involves the variables corresponding to arcs labeled by symbols representing calls to the entry (in the DFA of the calling task) and acceptances of calls at that entry (in the DFA of the accepting task).

The system of equations we have generated expresses a large part of the semantics of the constrained expression representing the behavior of the concurrent system. It states that a DFA representing a task must be in a single state at the start and end of the subsequence, that the number of times the DFA enters any other state must equal the number of times it leaves that state, and that the events corresponding to rendezvous of two tasks must occur the same number of times in the behavior of each of the two tasks. Any subsequence of an actual behavior starting with A and ending with B must satisfy the system of equations. Furthermore, if we were interested in subsequences satisfying some additional condition involving the occurrence of other events, we could easily augment the system of equations to express this. For example, if we wanted a bound on the duration of subsequences starting with A, ending with B, and containing no occurrences of event C, we would add equations stating that the arc variables labeled by the symbol corresponding to C must all be 0.

The total duration of a set of events corresponding to a solution of this system of equations is simply the sum of the arc variables, weighted by the durations of the events whose symbols label the arcs. Taking this weighted sum as the objective function, the integer programming component of our toolset will find a solution to the system of equations, assuming they are consistent,

giving the maximum possible value for the total duration. (If there is no maximum value, IMINOS will report that the duration is unbounded. In practice, for reasons discussed below, we usually impose some relatively large upper bound on the variables, thus ensuring that the duration is bounded.) This maximum possible duration reported by IMINOS is a bound on the total duration of the subsequences we are considering.

Because the system of equations represents only necessary conditions that must be satisfied by the subsequences of behaviors and does not completely characterize those subsequences, the solution found by the integer programming package may not correspond to a subsequence of a behavior. In that case, while the maximum value of the objective function found by the integer programming package is an upper bound on the durations of the subsequences in question, it need not be the least upper bound. There are two reasons that solutions to the system of equations may not correspond to subsequences of behaviors.

The first reason is that the equations represent most, but not all, of the semantics of constrained expressions. The equations do not guarantee that events will occur in the order required by the concurrent program, and it may be the case that this order is not consistent with the solution found by IMINOS. Furthermore, the system of equations does not completely constrain the number of occurrences of events labeling arcs forming a cycle in a task DFA. Consider a state i with an arc a from i to itself. The corresponding variable, x_a, will occur on both sides of the flow equation generated for i, since a will belong to both $\mathrm{In}(i)$ and $\mathrm{Out}(i)$. The flow equation thus does not restrict the value of x_a at all. Indeed, there may be solutions to the system of flow equations in which the variable x_a has a nonzero value but the variables corresponding to the other arcs entering and leaving state i are all zero. Such a solution cannot correspond to a subsequence of a behavior because it does not describe a path through each DFA. The equations saying that the number of calls from a task to an entry must equal the number of times those calls are accepted add enough additional restrictions to eliminate such solutions in many, but not all, cases.

The second reason that a solution to the equations may not correspond to a subsequence of a behavior is that it may not be possible to reach all of the start states in that solution at the same time in any actual behavior of the concurrent system. The equations do not impose any restrictions on the start states of the various DFAs, and therefore do not exclude solutions that are inconsistent with the behavior of the concurrent system.

In some cases, it is possible to tighten the bound obtained by integer linear programming through procedures that overcome some of the problems associated with cycles and with solutions whose initial states are not simultaneously reachable in an actual behavior. We now briefly describe a marking algorithm that reduces the problems due to cycles in DFAs and a procedure for generating additional equations to eliminate many solutions with unreachable initial states. Detailed descriptions of these methods are given in [5].

If upper bounds for all variables are introduced into the integer linear programming problem, the variables associated with cycles that can occur arbitrarily often will all take the maximum value in a solution to the equations giving an upper bound on durations. Such variables can be easily detected by inspection of the solution, and it might seem that a valid upper bound could be obtained by simply subtracting those variables from the solution. This is not the case, however, since the cycle may contain an event that occurs in the subsequence attaining the true maximum duration and eliminating the events in the cycle would eliminate this subsequence. Our marking algorithm removes certain cycles from the DFAs without eliminating any actual subsequences of behaviors. The idea is essentially to mark only those transitions in a task DFA that can be reached from a transition corresponding to the event starting the subsequences (or any other event that the analyst has indicated must occur in the subsequences). Any other transitions cannot occur in any subsequence starting with the specified event and can thus be removed (equivalently, one can think of setting the corresponding arc variables to zero).

The problem with unreachable start states can be addressed by introducing additional variables and generating equations representing conditions that must be satisfied by the initial segment of a behavior containing the subsequence of interest. Essentially, we use the equations to simultaneously find a subsequence starting with A and ending with B and an initial segment that would make that subsequence possible.

We have modified the constrained expression toolset to implement these methods and applied them to several examples with very encouraging results. The upper bounds found by the modified toolset are generally quite good, and frequently are indeed attained by subsequences of behaviors. Detailed discussion of the methods and an illustration of their application are given in [5].

CONCLUSIONS AND FUTURE DIRECTIONS

In this paper, we have given a brief, high-level description of the current status of our work on automated analysis of concurrent and real-time software. Results obtained from applying our automated constrained expression analysis techniques to a range of representative, nontrivial concurrent and real-time programming problems suggest that these techniques have the potential to be of significant value for realistic analysis applications. Our experiments have also suggested a number of directions for further improvements to the techniques and the tools that automate them. We outline a few of these here; more detailed discussions appear in [4] and [5].

The performance of our constrained expression toolset on a range of concurrent system analysis problems is already quite impressive. Various changes to the toolset's components could make its performance even better, however, and changes to its user interface could make it much easier and more convenient to use. We plan, for example, to improve the deriver by removing some of the minor restrictions that it currently imposes on CEDL programs, such as a prohibition on global variables, and by replacing its semantic analysis phase with a better and faster one. We plan to improve the toolset's integer programming component by introducing new and better branch-and-bound strategies making use of semantic information from the CEDL design in choosing a branching variable. We are also exploring alternative approaches to solving inequalities that can take advantage of the special structure of the inequality systems generated by our analysis techniques. We also expect to expand the capabilities of the behavior generator, such as its ability to help an analyst add inequalities that eliminate spurious ILP solutions, and to improve its heuristics so as to reduce its search times. Finally, we plan to add a uniform and friendly user interface, as well as improving the interfaces between the tools.

We also plan to extend the constrained expression formalism and analysis techniques so that we can use them to address a wider range of analysis problems. Among the topics that we are investigating are methods for directly handling more complex queries, such as "Can event A occur between event B and event C?", which are needed to directly analyze such logical properties as mutually exclusive use of resources. We are also looking at ways to express infinite behaviors, so that questions of fairness and starvation can be addressed by our analysis techniques. Another very important concern is with approaches to decomposing analysis problems into smaller parts, and then recombining the

results of analyzing those smaller parts in a way that gives accurate analyses of the full system. Finally, we are interested in developing both formal and empirical characterizations of the range of analysis problems and classes of concurrent systems to which constrained expression analysis techniques can fruitfully be applied.

Our application of the constrained expression analysis techniques and tools to real-time problems is still in the early stages. One immediate goal for our work in this area is an extensive experimental evaluation of our current automated analysis of timing properties, similar to the experimentation we have done with our automated analysis of logical properties of concurrent systems. We are also investigating several extensions to our real-time analysis techniques, including approaches to representing "truly concurrent" events, such as would occur in a multiprocessor or distributed real-time system, and methods for describing the effects of various scheduling mechanisms on real-time system behavior. Our preliminary efforts in both of these areas involve extensions to both the constrained expression formalism and the toolset, whose impacts have yet to be assessed.

Extensive experimentation with the constrained expression toolset has shown it to be very effective for performing automated analysis of logical properties of a range of concurrent system examples of realistic size and nontrivial complexity. Preliminary experimentation with applying the toolset to analysis of timing properties has yielded promising results. We expect that further experimentation and improvements along the lines suggested above will lead to automated analysis techniques for concurrent and real-time software that can contribute significantly to the robustness and reliability of this class of software systems.

Acknowledgements

The work described here has benefited immensely from our long-term collaboration with Dr. Laura Dillon. Ugo Buy and James Corbett have also made major contributions to this work. Susan Avery, Michael Greenberg, RenHung Hwang, G. Allyn Polk, and Peri Tarr have all contributed to the toolset implementation or experimentation activities described in this paper.

REFERENCES

[1] *Proceedings of the International Workshop on Real-Time Ada Issues,* sponsored by Ada UK and ACM SIGAda, May 1987. Appeared as *Ada Letters*, 7(6), Fall 1987.

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

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

[4] G. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. Submitted for publication. Available as Technical Report 90-116, Department of Computer and Information Science, University of Massachusetts, Amherst.

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

[6] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Constrained expression analysis of real-time systems. Technical Report 89-50, Department of Computer and Information Science, University of Massachusetts, Amherst, 1989.

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

[8] U. A. Buy. Solving integer programming problems using the IMINOS prototype. In preparation.

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

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

[11] J. C. Corbett and G. A. Polk. A tool for automatic generation of behaviors for constrained expression analysis. In preparation.

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

[13] L. K. Dillon. Overview of the constrained expression design language. Technical Report TRCS86-21, Department of Computer Science, University of California, Santa Barbara, October 1986.

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

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

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

[17] J. C. Wileden and G. S. Avrunin. Toward automating analysis support for developers of distributed software. In *Proceedings of the Eighth International Conference on Distributed Computing Systems*, pages 350–357. IEEE Computer Society Press, June 1988.