Experiments with an Improved Constrained Expression Toolset

George S. Avrunin* Ugo A. Buy* James C. Corbett* Laura K. Dillon† Jack C. Wileden*‡

Abstract

At TAV3, we described a preliminary version of the *constrained expression* toolset, and reported on the results of our initial experiments with it. Through those experiments we discovered shortcomings in some of the tools that limited the size of the examples that we could analyze. We have since redesigned and reimplemented several components of the toolset, with performance improvements of more than two orders of magnitude in some cases. The improved toolset has been successfully used with designs that involve hundreds of concurrent processes. In this paper, we describe several experiments with the new version of the toolset, including preliminary experiments with a technique for analyzing systems that include an essentially arbitrary number of identical components.

1 Introduction

At the 1989 Symposium on Software Testing, Analysis, and Verification, we described a preliminary version of a toolset for analyzing designs for concurrent systems, and reported on the results of some experiments with it [3]. The toolset automated the main *constrained expression* analysis techniques, and the results presented at the conference, which were somewhat better than those that appeared in the proceedings volume, were encouraging. Nonetheless, these results demonstrated some severe limitations in the toolset. For example, the toolset could not complete the analysis of a ten philosopher version of the standard dining philosophers problem.

Since that conference, we have redesigned and reimplemented several of the components of the toolset, leading to significant improvements in performance. For instance, the toolset now carries out a fully automated analysis of a 100 philosopher version of the dining philosophers problem (comprising 200 concurrent processes), beginning with source code in an Ada-based design language and producing an execution trace displaying deadlock, in approximately 20 minutes on a DECstation 3100. This and other experimental results indicate that the constrained expression analysis techniques can serve as a foundation for useful tools for software developers.

In this paper, we report on some experiments with the new implementation of the constrained expression toolset. We begin with a very brief overview of the constrained expression formalism and toolset in the next section. The third section describes some analyses carried out with the toolset, and the fourth section discusses some current and planned research intended to extend and improve our analysis techniques and the tools implementing them. The final section contains a summary and conclusions.

2 The Constrained Expression Toolset

2.1 Overview

The constrained expression formalism provides what is essentially a formal language approach to the description of

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

†Research partially supported by NSF grant CCR-8702905

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

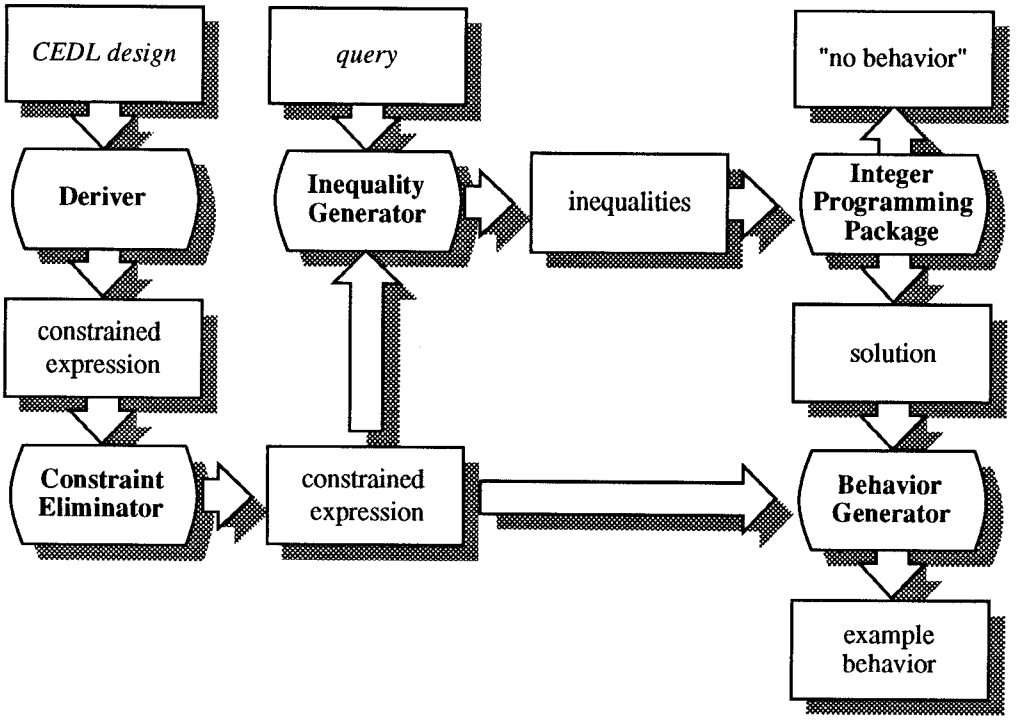

Figure 1: Diagram of Constrained Expression Toolset

executions of concurrent systems. The "unconstrained" behavior of each process in a concurrent system is represented by a regular expression, called a *process expression* (or alternatively, in accordance with Ada terminology, a *task expression*), over an alphabet of event symbols. An execution of the concurrent system being analyzed involves an execution of each of these processes, subject to additional restrictions such as those, for example, imposed by the communication and concurrency primitives of the language in which the system is implemented. We represent these restrictions by additional expressions, called *constraints*, which need not be regular. A string of event symbols thus represents the trace of an execution of the concurrent system if each of its projections on the alphabet of a process expression lies in the language of that expression, so that the string describes an execution of each process, and each of its projections on the alphabet of a constraint lies in the language of that constraint, so that the string satisfies the additional restrictions. (For simplicity, in this paper we will regard the executions of concurrent systems as represented by traces, and so as totally ordered in time. The formalism and our analysis techniques are, however, compatible with viewing the executions as corresponding only to partial orders of events [1].) Detailed and rigorous presentations of the formalism are given in [5] and in the appendix to [7], and less formal treatments intended to provide a more intuitive understanding of its features appear in [1] and [4].

The current version of the automated constrained expression analysis toolset is intended for use with concurrent system designs written in an Ada-like design language called CEDL (Constrained Expression Design Language) [6]. In normal use, an analyst would first use the *deriver* to produce a constrained expression representation from a concurrent system design written in the CEDL design language. The constrained expression produced by the deriver would then be used as input to the *constraint eliminator*, which intersects some of the task expressions corresponding to the tasks of the CEDL design and some of the constraints, producing an equivalent constrained expression with fewer constraints. (This procedure facilitates the analysis of systems in which the flow of control in a task depends on the manipulation of data.) 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 reflecting the constrained expression and the query. 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 (typically one that minimizes some measure of size). When a solution is found, the *behavior generator* uses this solution to guide a highly constrained, depth-first search to determine whether this solution corresponds to an actual system trace, and to produce such a trace if it does. The

discussion below describes some results of using the toolset to analyze concurrent system designs. More complete descriptions of the individual tools are given in [1] and [3]. The organization of the toolset is illustrated in Figure 1.

2.2 Improvements in the toolset

Four of the five components of the toolset — all except the deriver — have been extensively modified since TAV3, leading to significant improvements in performance. In this section, we briefly describe the changes.

In the results presented at TAV3, the constraint eliminator represented a major bottleneck. Part of this was due to the somewhat inefficient implementation of the prototype, but the chief difficulty involved the conversion of deterministic finite automata (DFAs) into regular expressions (REs). The constraint eliminator converts RE representations of task expressions and constraints into DFAs in order to carry out the intersection process, and the version of the inequality generator in use at the time expected to see RE representations. The constraint eliminator therefore converted the DFA obtained by intersecting a task expression and a constraint back into an RE. Unfortunately, the resulting RE may be very much larger than the DFA from which it is generated, and the process of producing the RE from the DFA may be quite expensive.

However, very efficient systems of inequalities can easily be generated from REs, where "efficient" is a measure of the relation between the size of the system of inequalities and the size of the object from which it is generated. We have therefore developed a hybrid representation for task expressions that we call *regular expression deterministic finite automata* (REDFAs). REDFAs are DFAs whose arcs are labeled with regular expressions satisfying certain conditions that preserve determinacy. Unlike REs, REDFAs are never significantly larger than the DFAs from which they are generated. Unlike DFAs, very efficient systems of inequalities can be easily generated from REDFAs.

We have therefore reimplemented the constraint eliminator, improving the efficiency of its code and modifying it so that the task expressions it outputs may be represented as REs, DFAs, or REDFAs, at the user's option. In general, we use the REDFA representation. These changes produced substantial improvements in the performance of the eliminator. The inequality generator was also modified to accept input in any of the three representations, and to write out additional semantic information for use by the integer programming package.

The integer programming package used for the experiments described in our paper in the TAV3 *Proceedings* had previously been installed at the University of Massachusetts as part of another project. We had encountered problems with it and, by the time we presented our results at TAV3, had implemented our own branch-and-bound integer programming system, IMINOS. IMINOS uses the MINOS optimization system [10] to solve the LP relaxations of IP problems. Very recently, we have modified the branching strategy used by IMINOS to take advantage of the semantic information produced by the inequality generator. This has resulted in substantial improvements in the performance of IMINOS on most of our examples.

The behavior generator has also been reimplemented to improve its efficiency and add some functionality. The new version can make use of more information contained in a solution to the system of inequalities, and provides more information to the user. It also has facilities that allow the analyst to use it in a more interactive fashion by using only a part (possibly none) of the solution and by modifying the solution to require or prohibit certain event sysmbols from occurring in the behavior string. Furthermore, it now incorporates a depth bound to its search through the global state space, and allows the analyst to extend that bound if no solution is found.

3 Performance of the Toolset

In this section we discuss a number of experiments with the constrained expression toolset, including new results for the examples discussed in [3]. Additional experiments are described in [1].

3.1 Dining philosophers

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

We have analyzed several variations of this system. In the basic one, 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. The fork task loops forever, accepting calls first at its `UP` entry and then at its `DOWN` entry. Each philosopher is represented by a task that repeatedly calls the `UP` entry of the fork to its "left", the `UP` entry of the fork to its "right", and then the `DOWN` entries of the two forks. A system with n philosophers thus has $2n$ tasks. Analysis is intended to detect the possibility of deadlock.

In Figure 2, we show the performance of the constrained

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	40	111	4	30	4	18	381×320	167
40	80	208	10	101	15	39	761×640	375
60	120	305	20	210	21	73	1141×960	629
80	160	413	34	336	34	109	1521×1280	926
100	200	522	52	479	44	153	1901×1600	1250

Figure 2: Toolset Performance on Basic Dining Philosophers Problem

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	41	140	121	231	249		603×1261	741
30	61	190	502	798	1100		903×2491	2590
40	81	265	1519	2370	3143		1203×4121	7297

Figure 3: Toolset Performance on the Dining Philosophers with Host

expression toolset on several sizes of the basic dining philosophers system, for all of which the toolset produced a system trace displaying deadlock. The columns give, 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 × 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. For comparison, note that the analysis of a version with eight philosophers reported in [3] took 234 seconds on a Sun 3/60, and the version of the toolset described in that paper was unable to complete the analysis of a version with ten philosophers.

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, the numbers of inequalities and variables and some of the tool execution times are much larger for the systems with host. This reflects the extra complexity introduced by the additional entry calls in the system with the host and by the fact that control flow in that system depends on the value of the variable representing the number of philosophers in the dining room. As a rough measure of this extra complexity, we note that the number of reachable states for the system with 40 philosophers and host is at least 100,000 times the number of reachable states for the basic system with 40 philosophers. Because the constrained expression approach does not require enumeration of all the reachable states of the system, however, the size of the corresponding system of inequalities and the tool execution times go up by much smaller factors.

Again, we analyzed these systems to detect possible deadlock due to all the philosophers picking up forks. Performance of the toolset for n dining philosophers with host is shown in Figure 3 for several values of n. The columns in the table show the number of philosophers, the number of tasks, the times for the various tools, the size of the system of inequalities, and the total time, just as in Figure 2. In each case, IMINOS reports that there is no integral solution to the system of inequalities, implying that no such deadlock is possible. It is therefore not necessary to run the behavior generator in these cases. As reported in [3], the previous version of the toolset required 687 seconds on a Sun 3/60 for a version with five philosophers. We note that these examples and the readers-writers system with writer priority described below are the only ones we have tried for which the new branching strategy for IMINOS actually leads to worse performance. The table shows the results with the new strategy for consistency with the other data presented in this paper; the IMINOS times with an earlier version of the toolset were 65, 58, and 81 seconds, respectively.

For comparison, we also analyzed the dining philosophers with host in the case where an incorrect guard on the host's

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
20	41	141	131	246	76	39	607×1305	633
30	61	196	491	862	107	104	905×2523	1760
40	81	259	1571	2509	265	200	1205×4163	4804

Figure 4: Toolset Performance on the Dining Philosophers with Erroneous Host

phils	tasks	deriv	elim	ineq	IMINOS	behav	size	total
5	11	59	2	13	3	11	144×174	88
5	6	49	89	74	5	11	303×585	228
5	6	51	30	7	1		73×95	89
5	5	36	1	5	1		96×80	43
5	5	49	7	12	2		192×202	70

Figure 5: Toolset Performance on Other Versions of the Dining Philosophers Problem

ENTER entry allows all the philosophers to enter the dining room at once. The performance of the toolset on these problems is shown in Figure 4. In each case, the toolset produced a behavior exhibiting the deadlock.

Several other versions of the dining philosophers problem have been investigated. We report briefly on the analysis of a few of these with the constrained expression toolset.

Young et al. [13] used their CATS system to analyze three-, four-, and five-philosopher examples of an "unrolled" version of the dining philosophers with host. In this version, the host task does not use a variable to keep track of the number of philosophers in the dining room, but instead uses nested `select` statements. The CATS system was used to verify a temporal logic assertion (that, under the assumption of a fair scheduler, each philosopher can get into the dining room). We used the constrained expression toolset to analyze the five-philosopher system for deadlock. The design published in [13] is not equivalent to the one in which the host uses a variable to keep track of the number of philosophers in the room (as was pointed out to us by Sol Shatz), and the constrained expression toolset correctly detects a possible deadlock in the "unrolled" version.

Karam and Buhr [9] analyze several versions of the dining philosophers problem for deadlock and starvation. Their systems use a single fork manager task to model the forks, rather than individual tasks. We analyzed CEDL versions of two of these systems for deadlock.

We also analyzed two modifications of the basic dining philosophers system that avoid deadlock without introducing other tasks. In the first of these, one philosopher picks up the forks in the reverse of the order used by the other philosophers, picking up, say, the fork on the right first while the others pick up the one on the left first. In the second version, which was suggested to us by Robert Kurshan, the philosophers pass around a "dictionary" token. A philosopher does not attempt to eat while holding the token, thereby ensuring that all the philosophers do not hold forks at once.

Results for these other versions of the dining philosophers problem are shown in Figure 5.The first row of the table gives the results for the five-philosopher "unrolled" system of [13]. In this case, the toolset produces a system trace displaying the deadlock. The second row gives results for a system with a fork manager in which deadlock is possible, and the third row gives results for a system in which the fork manager prevents deadlock by requiring the philosophers to pick up both forks at the same time. In the first of these latter two cases, the toolset produces a system trace displaying deadlock. In the second, IMINOS reports that no deadlock is possible, and it is not necessary to run the behavior generator. The fourth row of the table gives the results for the system in which one philosopher picks up the forks in the opposite order, and the fifth row gives the results for the system with the dictionary. IMINOS reports that deadlock is impossible in both of these cases as well, and the behavior generator is not run.

3.2 Gas station

The automated gas station example introduced by Helmbold and Luckham [8] has been studied by a number of authors (e.g., [9], [12]). This system models an automated gas station with an operator, a number of pumps, and a collection of customers. A customer pays the operator, who then activates a pump as appropriate. The customer then pumps gas, and the pump informs the operator of the amount pumped. The operator then gives change to the customer. Helmbold and Luckham used a series of iterative refinements of this system to illustrate their run-time monitoring system for debugging Ada tasking programs. Their examples involved

cus	tasks	deriv	elim	ineq	IMINOS	behav	size	total
2	4	36	30	15	8	6	120×200	95
2	4	37	34	16	6		125×209	93
3	5	44	2807	308	86	153	604×1401	3398
3	5	46	730	89	21	30	315×643	916

Figure 6: Toolset Performance on the Gas Station

systems with ten customers and three pumps.

We have analyzed several versions of the system that correspond to some of the refinements used by Helmbold and Luckham. The first of our systems contains two customer tasks, one pump task, and one operator task. Since we are interested in the concurrent aspects of the design, rather than the details of the computations performed by the various tasks, we ignore the amount of money paid by customers and the amount of change received. In this version, the operator does not activate the pump for a waiting customer until change has been given to the other customer. Because of a race between two customers who have both prepaid, the operator may attempt to give change to a customer who has not yet pumped gas, leading to deadlock. Our analysis is intended to detect this deadlock. In a second version, again with two customers, the operator activates the pump for any waiting customer before giving change. In this case, such a deadlock is impossible, and the toolset reports this. (Note that, even though deadlock is avoided, it is still possible for a customer to receive another customer's change. Karam and Buhr's [9] critical race assistant points up this possibility.)

When this deadlock-free two-customer design is scaled up to three customers, however, a more complicated race condition arises, again leading to the possibility of deadlock. (This was first noticed by K. C. Tai [11], who used a graphical analysis method to detect the error.) We analyzed two versions of such a three-customer gas station. The first is a straightforward extension of the two-customer design. In this case, the analysis must reflect the much larger number of possible states of the queue of waiting customers, leading to the long times for the constraint eliminator and the inequality generator and the relatively large system of inequalities. In the second version, the number of possible states is reduced by setting the slots in the queue to some fixed value when they are not occupied by customers 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 gas station examples are shown in Figure 6. The first column in the table shows the number of customers for each problem; the other columns are the same as in the preceding figures. The first line of the table gives results for the original two-customer gas station, in which deadlock occurs, and the second line gives information for the revised two-customer version without deadlock. The Sun 3/60 times we reported at the TAV3 symposium for these problems were 2339 and 1858 seconds respectively. (At the time we prepared our paper for the proceedings of the symposium, the toolset was not able to complete the analysis of these cases.)

Results for the three-customer extension are shown in the third line of the table, and those for the version that reduces the number of queue states 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 in the operator task.

One way to avoid deadlock and ensure that customers receive their own change is to have separate entries in the operator and pump tasks to distinguish the calls from various customers. In this variant of the system, the operator task maintains a flag for each customer indicating whether that customer has prepaid and is waiting for change. Our analysis of this variant of the gas station system was intended to determine whether a customer who has prepaid can be permanently blocked before pumping gas. The toolset correctly determines that this cannot occur.

For comparison, we also analyzed a version with two customers with an error (similar to that in the two-customer version discussed previously) that permits deadlock to occur. Results for the correct and incorrect versions of these systems are shown in Figure 7. The first row gives results for the erroneous two-customer version. The next five rows give the results for the correct versions. It is not necessary to use the behavior generator in the latter cases.

3.3 Readers and writers

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

cus	tasks	deriv	elim	ineq	IMINOS	behav	size	total
2	4	39	3	5	1	3	76×80	51
2	4	36	3	4	1		65×63	44
3	5	47	11	9	4		107×131	71
4	6	62	62	23	13		181×279	160
5	7	92	415	85	42		327×611	634
6	8	168	2991	308	180		633×1359	3647

Figure 7: Toolset Performance on Gas Station with Separate Entries for the Customers

(r,w)	tasks	deriv	elim	ineq	IMINOS	behav	size	total
(4,1)	6	40	7	9	6	3	82×137	65
(4,1)	6	40	5	4				49
(4,1)	6	41	9	9	273		90×148	332

Figure 8: Toolset Performance on Readers and Writers Problem

a writer and one or more readers could gain access to the resource at the same time.

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

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

Results for a few versions of these readers and writers systems are shown in Figure 8. The first column of the table contains an ordered pair giving the number of readers and the number of writers in the system. The first line of the table gives times for an incorrect system with four readers and one writer. In this system, an error in the controller task allows a deadlock. The second line gives results for a correct system that is analyzed for undesirable simultaneous access to the resource. In this case, the constraint eliminator removes that part of the controller process 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 table is just the time required to determine that the symbol does not occur in the constrained expression produced by the constraint eliminator. The times reported in [3] for 2-reader versions of these systems were 755 seconds and 1924 seconds, respectively. The third line in Figure 8 gives results for a system in which the controller gives the writer priority by accepting a call at START_WRITE at any time, but then disabling the entry START_READ and waiting for all readers who have access to the resource to relinquish it before allowing the writer to proceed. This system, which is correct, was analyzed to detect deadlock. As noted above, the new branching strategy for IMINOS leads to worse performance with this example; the IMINOS time with an earlier version of the toolset was 59 seconds.

4 Current Research

We are currently investigating a number of extensions to the constrained expression analysis techniques and modifications to the toolset to support those extensions and improve its performance. We briefly mention some of these ideas.

We have begun to develop methods for analyzing systems

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

Figure 9: Toolset Performance with Many Identical Tasks

that include an essentially arbitrary number of identical tasks and we have started modifying the toolset to support these methods. In conjunction with these techniques, 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 needs further modification for use with these two techniques, but we present in Figure 9 some results of applying the other components of the toolset to a system involving two coupled resource managers controlling two resources and a large number of identical customers who require equal amounts of each resource.

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

Having demonstrated that automated constrained expression analysis can be successfully applied to realistic sized problems of certain classes, and in some cases to arbitrarily large systems, we now wish to extend its application to additional classes of problems. One such class is real-time systems problems, specifically the analysis of timing properties of concurrent systems. We have recently extended our techniques and modified the toolset to carry out an automatic derivation of an upper bound on the time that can elapse between the occurrences of any two designated events in an execution of a logically concurrent system running on a single processor [2]. We are currently investigating methods for obtaining similar results for the situation in which each process in the concurrent system runs on its own processor (the case of "maximal parallelism") and we intend to examine the more difficult multi-processor case in which there are more processes than processors. We are also studying ways to take particular scheduling disciplines into account in our analysis.

We would also like to extend our techniques to apply to so-called "fairness" questions, such as whether one or more processes in a concurrent system can "starve", i.e., wait indefinitely for a resource that is repeatedly available but perpetually given to other processes. Analyzing this class of questions will require extending the constrained expression formalism to represent infinite behaviors. Another class of interest is questions that are expressed in terms of the order in which events occur, such as questions concerning mutual exclusion. Although we have successfully used our existing methods to answer some questions of this type, such as the readers and writers problems, our existing methods are not able to address them directly, and we must modify the system being analyzed in order to carry out our analysis. Some techniques for handling segments of executions, which we have developed for use in analysis of real-time systems problems, will provide a starting point for tackling this class of questions.

Another important research direction under investigation is the modularization of constrained expression representations and of their analysis. The current toolset analyzes complete, self-contained systems. In order to support analysis of individual system components, we propose extending constrained expressions with *environment constraints* that express assumptions about the environment in which a component executes. We are currently experimenting with this approach and with methods for composing the constrained expression representations of system components. Such methods will make it possible to apply constrained expression analysis to incomplete system designs and to designs for still larger, more complex systems.

An additional area of future research is the solution of the integer linear systems we generate. We chose to base the integer programming component of the toolset on MINOS for

several reasons, including the availability and robustness of MINOS and the relative ease of adding the branch-and-bound mechanism to it. The performance of IMINOS significantly improves on the results obtained from the Land and Powell package, which was used in the experiments described in [3]. While the performance of IMINOS, particularly with the new branching strategy described earlier, has been very satisfactory for demonstrating the feasibility of our general approach, further development of the toolset would benefit from improved integer programming methods. We are therefore continuing to experiment with refinements to our branching strategy and also investigating special-purpose algorithms for the solution of the network flow problems with side constraints that are generated by our method.

5 Summary and Conclusions

The redesign and reimplementation of several components of the constrained expression toolset have resulted in dramatic performance improvements on a range of example analysis problems. Perhaps most strikingly, the improved 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 approximately 20 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 involving hundreds of concurrent processes. This is in marked contrast to the results reported for most other methods that have been implemented, notably those based on constructing and searching a reachability tree.

The results of our most recent experiments indicate the potential value of the constrained expression approach. 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. 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. S. Avrunin, U. A. Buy, J. C. Corbett, L. K. Dillon, and J. C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Trans. Softw. Eng.*, to appear.

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

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

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

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

[6] 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.

[7] 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.

[8] D. Helmbold and D. Luckham. Debugging Ada tasking programs. *IEEE Software*, 2(2):47–57, March 1985.

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

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

[11] K. C. Tai. A graphical notation for describing executions of concurrent Ada programs. *Ada Letters*, 6(1):94–103, January–February 1986.

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

[13] M. Young, R. N. Taylor, K. Forester, and D. Brodbeck. Integrated concurrency analysis in a software development environment. In R. A. Kemmerer, editor, *Proceedings of the ACM SIGSOFT '89 Third Symposium on*

Software Testing, Analysis and Verification, pages 200–209, 1989. Appeared as *Software Engineering Notes*, 14(8).

(G. S. Avrunin) DEPARTMENT OF MATHEMATICS AND STATISTICS, UNIVERSITY OF MASSACHUSETTS AT AMHERST, AMHERST, MA 01003
E-mail: avrunin@math.umass.edu

(U. A. Buy) DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS AT CHICAGO, BOX 4348, CHICAGO, IL 60680
E-mail: buy@figaro.eecs.uic.edu

(J. C. Corbett and J. C. Wileden) DEPARTMENT OF COMPUTER AND INFORMATION SCIENCE, UNIVERSITY OF MASSACHUSETTS AT AMHERST, AMHERST, MA 01003
E-mail (Corbett): corbett@cs.umass.edu
E-mail (Wileden): jack@cs.umass.edu

(L. K. Dillon) DEPARTMENT OF COMPUTER SCIENCE, UNIVERSITY OF CALIFORNIA AT SANTA BARBARA, SANTA BARBARA, CA 93106
E-mail: dillon%cs@hub.ucsb.edu