Experiments with Automated Constrained Expression Analysis of Concurrent Software Systems

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

Laura K. Dillon†
Department of Computer Science
University of California, Santa Barbara

Jack C. Wileden*,‡
Department of Computer and Information Science
University of Massachusetts, Amherst

Abstract

It is unlikely that any single approach to analysis of concurrent software systems will meet all the needs of software developers throughout the development process. Thus, experimental evaluation of different analysis techniques is needed to determine their relative strengths and practical limitations. Such evaluation requires automated tools implementing the analysis techniques.

This paper describes a prototype toolset automating the constrained expression approach to the analysis of concurrent software systems. The results of preliminary experiments with the toolset are reported and the implications of these experiments are discussed.

1 Introduction

A wide variety of techniques have been proposed for analyzing the behavior of concurrent software systems. These differ in their underlying models of concurrent computation, in the questions about behavior they attempt to answer, and in the stages of the software development process in which they are applied. It is unlikely that any single approach to analysis can possibly meet all the needs of software developers throughout the development process.

The effective use of analysis techniques during software development requires an understanding of their relative strengths and practical limitations. While virtually all existing analysis techniques are known to have limitations of various kinds, little is known about the practical significance of these limitations. This determination can only be made through experimental application of the techniques to a wide range of concurrent systems. Clearly, experiments must be conducted with systems of realistic size and complexity, and so automated tools implementing the analysis techniques will be required. In this paper, we report on a prototype toolset supporting the *constrained expression* approach to analysis, and the results of some preliminary experiments with that toolset.

The next section of the paper briefly describes the constrained expression approach. The third section describes the toolset, and the fourth reports some of our experience in using the toolset. Finally, we discuss the implications of these experiments for further work on constrained expressions.

2 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. A detailed and rigorous presentation

---

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

†Partially supported by NSF grant CCR-8702905

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

is given in [11], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [5]. The use of constrained expressions with a variety of development notations is illustrated in [5] and [13].

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.

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 the system 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 [13].

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 starvation of component processes 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 inequalities 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.

Constrained expression analysis, then, is a static, event-based approach (though the construction of a behavior from a solution of a system of inequalities has similarities to dynamic analysis). The constrained expression formalism is closely related to path expressions [7], event expressions [21], and COSY [20]. More detailed discussion of the relation between constrained expressions and a variety of methods for describing and analyzing concurrent software systems can be found in [5] and [24]. The constrained expression analysis techniques can be regarded as rigorous formulations of methods based on arguments about the order and number of occurrences of events. Such methods have been widely used in conjunction with concurrent software systems (e.g., [16]).

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. One potential difficulty in applying the approach is that it requires that analysts correctly formulate questions about the behavior of a system in terms of patterns of event symbols in strings representing system behaviors. Other potential drawbacks include the difficulty of automating some aspects of generating and reasoning about the systems of inequalities.

After manually applying the constrained expression analysis techniques to a number of small examples with encouraging results (e.g., [2], [5], [6], [24]), we began to construct prototype tools automating various aspects of the analysis. An important goal of this automation effort is to support experimentation directed at determining the practical significance of the potential problems cited above. This paper describes the first complete version of the prototype toolset and reports the results of some experiments with it.

3 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 and the solution found by the integer programming package (when the inequalities are consistent) 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.

The current toolset is intended for use with designs written in the Ada-based design language CEDL (Constrained Expression Design Language) [12]. 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 the Ada control-flow constructs have correspondents in CEDL. We have chosen 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 [23]. Those aspects of the toolset that depend on CEDL are noted below.

Examples showing the manual application of the analysis techniques automated by the prototype toolset have appeared previously. In particular, a CEDL formulation of the standard dining philosophers problem, the corresponding constrained expressions, simplified versions of those expressions, an outline of the inequality generation and solution process for those expressions, and an example behavior corresponding to the deadlock discovered by that process appear in [5]. Similarly, parts of the CEDL formulation of Helmbold and Luckham's gas station problem [17], parts of the corresponding constrained expressions and some of the inequalities generated in analyzing that problem appear in [24]. The intermediate inputs to and outputs from the tools

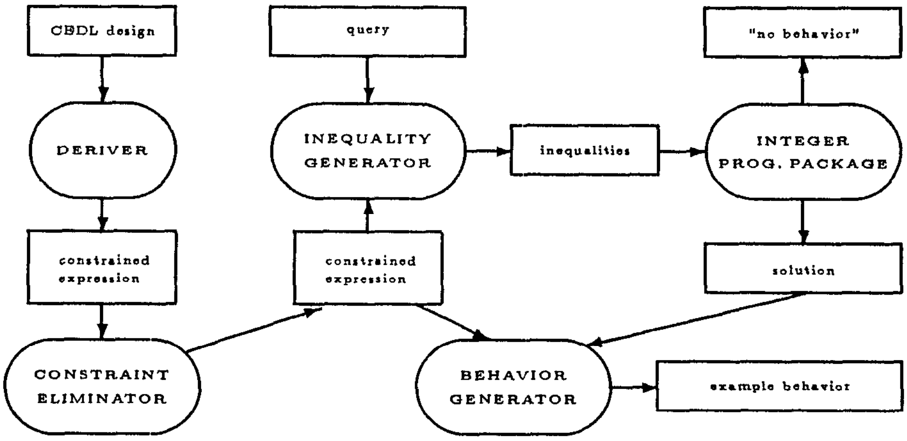

Figure 1: Diagram of Constrained Expression Toolset

in our toolset are not designed to be easily human readable. Hence, we do not include examples of them in the descriptions of the individual tools presented below, but refer the reader to the corresponding information in [5] and [24].

The deriver [1] produces constrained expression representations from CEDL system designs. It is written in Ada, and was developed using Arcadia-produced versions of standard compiler construction tools and the Graph Definition Language and GRAPHITE processor [9]. The deriver generates a graph representing the constrained expression. Eventually, this will be the standard internal representation for constrained expressions. Currently, however, the prototypes of the other tools expect input in other formats, and small utility programs convert between the formats. For a CEDL design, the system expression of the constrained expression representation produced by the deriver consists of the interleave of *task expressions* representing the behavior of the tasks in the system. The deriver also generates all required constraints.

The constraint eliminator [14] is written in Common LISP, and shares a common front end with the behavior generator. It takes a "generator expression", which is a subexpression of the system expression, and constraints involving symbols from the generator expression, and produces a new expression whose language is the set of strings in the language of the generator expression that satisfy the constraints. The constraint eliminator converts the generator expression and constraints into finite state automata (the constraints in a constrained expression representation of a CEDL system are regular), from which it produces a new automaton accepting the intersection of the appropriate languages. It then returns an expression corresponding to this automaton. In principle, the generator expression need only be regular, and so could be the full system expression. However, the process of intersecting the finite state automata quickly becomes intractable if the generator expression involves the interleave operator. For this reason the current constraint eliminator will only accept generator expressions that use the standard regular expression operators. When analyzing constrained expressions derived from CEDL designs, the constraint eliminator is typically used with a task expression and the constraints that enforce correct dataflow within tasks. The input task expression is then replaced with the expression returned by the constraint eliminator and the intra-task data flow constraints are eliminated. This process facilitates certain aspects of the analysis of the constrained expression, as indicated below.

The inequality generator [3], which is also written in Common LISP, takes a constrained expression representation in essentially the same format as that accepted by the constraint eliminator, and generates a system of linear inequalities representing part of the semantics of the constrained expression. The inequality generator builds an abstract syntax tree representing each task expression, and generates inequalities based on the semantics of regular expressions. It then generates additional inequalities derived from some of the constraints. The full system of inequalities involves both the total numbers of occurrences of various event symbols and the numbers of times various branches in the abstract syntax trees are traversed. However, these inequalities do not reflect the *complete* semantics of the constrained expression. For example, not all the information about the relative order of event symbols is represented, so that the constraints that enforce correct dataflow are not reflected in the generated system of inequalities. (The significance of this problem for intra-task dataflow is reduced by application of the constraint eliminator.) In addition, the generated system of inequalities does not completely reflect the semantics of the alternation operator when one of its operands is the Kleene star of an expression. The full semantics would require quadratic inequalities, and the integer programming package we are currently using only handles linear systems. As mentioned below, we are currently investigating another integer programming package that may eliminate this problem.

The inequality generator also provides an interactive facility allowing the analyst to add inequalities representing assumptions or queries about the behavior of the system. The inequality generator produces an output file giving the system of inequalities in the format required by the integer programming package, as well as a human-readable report giving the correspondence between variables in the system of inequalities and event symbols. If the integer programming package finds a solution to the system of inequalities, the inequality generator uses this correspondence to report the solution to the analyst in terms of traversal of the abstract syntax trees and the numbers of occurrences of event symbols. Certain aspects of the inequality generator, including the representation of various constraints, depend on features of CEDL. Its basic structure, however, is compatible with all constrained expressions.

The integer programming package that we are currently using is a branch-and-bound integer linear programming system [19] written in FORTRAN; it was chosen because it had already been installed as part of a previous project at the University of Massachusetts. We have encountered some problems with its branch-and-bound strategy, as described in the next section, and with the limitation to linear sys-

system	deriver	constraint eliminator	inequality generator	int. prog. package	behavior generator	total CPU time
DP-3	74	—	11	4	19	108
DP-4	82	—	13	5	26	126
DP-5	94	—	15	6	32	147
DP-6	109	—	18	9	38	174
DP-8	142	—	24	14	54	234
DP-10	177	—	30	—	—	—
DPH-3	123	77	23	7	—	230
DPH-4	133	194	62	41	—	430
DPH-5	152	330	102	103	—	687
RW-I	43	539	32	17	124	755
RW-C	63	1740	38	83	—	1924
GAS-I	75	960	220	—	—	—
GAS-C	76	944	190	—	—	—

Figure 2: Sun 3/60 CPU times, in seconds, for the constrained expression tools.

tems. We are currently implementing integer programming on top of the MINOS optimization package [22].

The behavior generator [15] is a Common LISP program for producing system behaviors with certain properties. Input to the program consists of a constrained expression and counts for certain event symbols (counts produced by the integer programming package). The behavior generator builds finite state automata corresponding to the task expressions and constraints of the constrained expression. It then uses heuristic search techniques to find a string of event symbols representing a system behavior with the given numbers of symbol occurrences. It can be used with any constrained expression having regular constraints.

4 Using the Toolset

We have begun to use the prototype toolset in the analysis of concurrent systems. The preliminary experiments reported below represent an initial attempt to determine the practical limitations of automated support for the constrained expression approach to analysis. A number of variations of four different systems are analyzed. First, a standard formulation of the dining philosophers problem provides a basis for comparison with other analysis techniques because of its widespread use as a benchmark problem. The addition of a host task controlling entry to the dining room indicates how the introduction of intra-task dataflow affects tool performance. The readers/writers problem requires analysis of more complex data flow patterns. Finally, we analyze an automated gas station example in which both the synchronization patterns and intra-task dataflow are relatively complex. Varying the number of philosophers in the dining philosophers problems indicates how increasing the number of tasks in the system being analyzed affects tool performance. We consider both time and space efficiency of the tools.

The table in Figure 2 gives CPU times for the application of the components of the toolset to these systems. The table in Figure 3 provides information about the sizes of the constrained expression representations and the system of inequalities used in the analysis for each case.

The first six rows of the tables give the data for versions of the standard dining philosophers problem with three, four, five, six, eight, and ten philosophers, respectively. In the analysis reported here, we seek to determine whether a particular philosopher task could possibly wait indefinitely for a rendezvous with a second fork task, and thus starve (in both the concurrent systems and metaphorical senses). These systems do not have a doorkeeper or host to prevent all the philosophers from trying to pick up forks at the same time, and are therefore subject to deadlock in which each philosopher task starves waiting to rendezvous with a second fork task. The dining philosophers systems without host use rendezvous simply for synchronization purposes, and involve no intra-task dataflow. We therefore do not use the constraint eliminator in these cases.

The size of the constrained expression representations of these systems goes up linearly with the number of philosophers, as does the execution time of the inequality generator and the size of the system of inequalities generated. We have successfully applied the deriver and inequality generator with systems containing up to twenty philosophers (i.e., forty concurrent tasks), and expect no difficulties with even larger systems. However, the integer programming package we are currently using is unable to solve the systems of inequalities generated in the cases with more than eight philosophers, due to failure of an accuracy test in the course of solving a linear programming relaxation of the integer linear programming problem. We discuss the implications of this failure below. In the cases where a solution to the system of inequalities is found, the behavior generator produces a behavior exhibiting the deadlock.

The next three rows of the tables give data for analyses of three-, four- and five-philosopher versions that have a host task to prevent all the philosophers from entering the dining room and trying to pick up forks at the same time. Again, the analysis seeks to determine whether a particular philosopher can starve. In these cases, the constraint eliminator is applied to the task expression for the host, along with the constraints enforcing consistent use of the variable that counts the number of philosophers in the dining room. The resulting task expression is used in the input to the inequality generator. Because this task expression must represent the effects of all possible execution paths on the variable that counts philosophers in the dining room, the size of the constrained expression used as input to the inequality generator and the size of the resulting system of inequalities both go up rapidly with the number of philosophers, and are significantly greater for the five-philosopher system with a host than for the eight-philosopher system without a host. Due to the detailed structure of the particular system of inequalities, however, the integer programming package does not encounter accuracy problems here, and, in each of the three cases, reports that no philosopher starves. Thus, it is not necessary to use the behavior generator in these cases.

The tenth and eleventh lines of the tables give data for two CEDL versions of the readers/writers examples of [8]. These

	output of deriver		output of eliminator		output of inequality gen.	
system	operators	symbols	operators	symbols	inequalities	variables
DP-3	69	138	—	—	58	48
DP-4	92	184	—	—	77	64
DP-5	115	230	—	—	96	80
DP-6	138	276	—	—	115	96
DP-8	184	368	—	—	153	128
DP-10	230	460	—	—	191	160
DPH-3	133	289	187	383	96	121
DPH-4	168	385	658	1276	202	483
DPH-5	203	481	1169	2206	295	905
RW-I	133	253	451	768	117	243
RW-C	199	360	501	1082	130	281
GAS-I	320	571	3048	7539	804	1751
GAS-C	146	385	2620	6666	690	1494

Figure 3: Sizes of constrained expressions and systems of inequalities. The second and third columns give the numbers of regular expression operators and event symbols for the constrained expression produced by the deriver; the fourth and fifth columns give the corresponding figures for the constrained expression produced by the constraint eliminator. The last two columns give the numbers of inequalities and variables in the system produced by the inequality generator.

systems involve four tasks: two readers, one writer,and a controller that is intended to prevent a reader and a writer from simultaneously using the shared data. The first system is an incorrect implementation in which the value of a variable indicating that a writer has access to the shared data is set incorrectly when the writer finishes with the data. In this case, the analysis determines that the tasks in the system can starve waiting for access to the data. The second system is a correct implementation, and manual analysis of the constrained expression representation establishes this fact. The toolset, however, cannot directly answer questions involving the occurrence of one event between two others, such as "Does a reader attempt to use the shared data between the writer gaining and releasing access to it?" This is because the inequalities produced by the inequality generator do not reflect all the information about the relative order of events. We therefore modify the CEDL code slightly to include an explicit test for the failure of mutual exclusion, and use the toolset to determine whether a corresponding error flag is ever set. The increased complexity of control flow caused by this modification accounts, at least in part, for the long execution time of the constraint eliminator in this case.

The last two lines of the tables give information for two CEDL versions of the automated gas station examples of [17]. These systems also involve four tasks: two customers, a pump, and an operator. The first system is an incorrect implementation in which deadlock can occur, and the second is a correct version. Analysis here is intended to determine whether starvation of customer tasks is possible. In these cases, we use the constraint eliminator with the task expression representing the operator of the automated gas station and the constraints enforcing consistent use of the variables that count and maintain a queue of the customers pumping gas. Even with only two customer tasks in the system, the behavior of the operator is considerably more complex than that of the host in the dining philosophers systems or the controller in the readers/writers systems, and the system of inequalities produced by the inequality generator is too large for the Land-Powell integer programming package, which allows a maximum of 999 variables. We expect that improvements in the constraint eliminator and the conversion to an integer programming package based on MINOS will very soon allow us to apply the complete toolset to these systems as well.

5 Conclusions

These initial experiments with the prototype constrained expression toolset are encouraging. The toolset provides complete analysis of both versions of the dining philosophers problem, with and without a doorkeeper. It also provides complete analysis of the readers/writers problem, although a minor alteration to the problem is required to make this automated analysis possible. Even the prototype versions of the tools are efficient enough to be useful to software developers on examples of moderate size. Furthermore, earlier experiments show that the constrained expression approach can detect a variety of errors and can be used with a broad range of design notations and programming languages.

However, some weaknesses of the prototype toolset are evident. The most significant of these involve the branch-and-bound integer programming package we are currently using [19], and include the limitations on the size of system of inequalities that the package can handle, the accuracy problems noted in the previous section, and the restriction to linear inequalities. This package is an implementation in FORTRAN 66 of the first branch-and-bound algorithm for general integer programs [18]. Its division scheme has been replaced, in virtually all commercial integer programming codes, by the variable dichotomy scheme first proposed by Dakin [10], and we believe that some of its strategies for selecting a branching variable and for exploring the tree may be poorly suited to our systems of inequalities. For these reasons and others, including the ability to handle quadratic inequalities, we expect a considerable improvement in performance from the integer programming package we are currently implementing using the MINOS optimization package [22].

Other drawbacks of the prototype toolset were not as significant in the experiments described here, but may become more important when the toolset is applied to a wider range of concurrent systems. These include the facts that the system of inequalities produced by the inequality generator does not reflect the full semantics of the constrained expression representation (though the use of quadratic inequalities with the MINOS system addresses part of this issue) and that the task expressions returned by the constraint eliminator may lead to larger systems of inequalities than other, equivalent expressions.

We are now beginning to address these issues, and a number of improvements to the toolset are planned. In addition to replacing the Land-Powell integer linear programming package with one based on MINOS that will allow quadratic inequalities, we intend to modify the behavior generator to use all the information contained in a solution to the system of inequalities, rather than just the total numbers of occurrences of the various event symbols. We will be modifying the inequality generator to produce the quadratic inequalities needed to express the semantics of the alternation operator when one of its operands is the Kleene star of an expression, and are investigating other ways to improve the generation of inequalities so that they reflect more of the full semantics of constrained expressions. We are also investigating several approaches to improving the performance of the constraint eliminator.

In addition, improvements to the interfaces between the human analyst and the toolset and between the tools are underway. Clearly, analysts should be able to formulate behavioral queries in terms of elements from the original system description and at a higher level of abstraction than is currently possible, and a common internal representation would help in integrating the various tools. We are currently experimenting with using the UTM-0 automated generator [25], a prototype tool supporting specification level interoperability, to provide access to a common internal representation for constrained expressions from both Ada and Common LISP programs. This will eliminate the need for the utility programs that now translate between the different constrained expression formats used by the various tools. It will also allow us greater freedom in choosing which languages to use in future implementations of improved versions of the various toolset components.

While starting to improve the prototype toolset, we have also begun to explore additional applications for constrained expression analysis, some of which may lead to enhancements to the underlying formalism and further modifications to the tools. In particular, we have begun to study the application of the constrained expression approach to various scheduling and real-time problems [4]. Because expressing some of these scheduling and timing problems, as well as the semantics of certain programming languages for concurrent systems, involves constraints that are not regular expressions, we hope to be able to eliminate the regularity restrictions in some of the tools.

For a more complete understanding of the strengths and weaknesses of the constrained expression approach and the prototype toolset, we need to evaluate the performance of the toolset on a wider range of examples. The problem of designing an appropriate suite of benchmark problems for concurrent software analysis tools has not been carefully studied; we hope to develop some criteria for such a suite in the course of collecting additional examples for experiments with the constrained expression toolset. It is unlikely that a single approach to analysis will meet the needs of developers of concurrent software, and such a test suite would be of significant value in comparing various approaches and determining the types of problems for which each approach has the greatest value.

Based on the prototype toolset and the initial experiments described in this paper, we are very encouraged about the prospective value of the constrained expression approach to automated analysis of concurrent software systems. We therefore plan to pursue the toolset improvements, enhancements to the formalism, and more extensive experimental evaluation outlined above. We expect that these activities, in conjunction with similar experimental evaluations by other researchers developing other analysis techniques, preferably all based on a common test suite, will result in improved understanding of the relative strengths and weaknesses of the constrained expression approach and alternative concurrent system analysis techniques.

Acknowledgments

Our students Susan Avery, Ugo Buy, Michael Greenberg, RenHung Hwang, and George Walden have played important roles in the development of the toolset and the experimental evaluation reported in this paper.

References

[1] 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.

[2] G. S. Avrunin. Experiments in constrained expression analysis. Technical Report 87-125, Department of Computer and Information Science, University of Massachusetts, Amherst, November 1987.

[3] G. S. Avrunin and U. Buy. An inequality generator for constrained expression analysis. In preparation.

[4] 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, 1989.

[5] 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.

[6] G. S. Avrunin and J. C. Wileden. Describing and analyzing distributed software system designs. *ACM Trans. Prog. Lang. Syst.*, 7(3):380–403, July 1985.

[7] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. In E. Gelenbe and C. Kaiser, editors, *Operating Systems*, volume 16 of *Lecture Notes in Computer Science*, pages 89–102. Springer-Verlag, Heidelberg, 1974.

[8] R. H. Carver and K.-C. Tai. Detection of synchronization errors in concurrent software by semantics-based analysis. Preprint, 1988.

[9] L. A. Clarke, J. C. Wileden, and A. L. Wolf. GRAPHITE: A meta-tool for Ada environment development. In *Proceedings of 2nd International Conference on Ada Applications and Environments*, pages 81–90, April 1986.

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

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

[12] 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.

[13] 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.

[14] L. K. Dillon and G. Walden. A prototype constraint eliminator for constrained expression representations. In preparation.

[15] M. Greenberg and S. Avery. A behavior generator. Software Development Laboratory Memo 89-1, Department of Computer and Information Science, University of Massachusetts, 1988.

[16] A. N. Habermann. Synchronization of communicating processes. *Commun. ACM*, 15(3):171–176, 1972.

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

[18] A. H. Land and A. G. Doig. An automatic method for solving discrete programming problems. *Econometrica*, 28:497–520, 1960.

[19] A. H. Land and S. Powell. *Fortran Codes for Mathematical Programming: Linear, Quadratic and Discrete.* John Wiley & Sons, Ltd., London, 1973.

[20] P. Lauer, P. Torrigiani, and M. Shields. COSY: A system specification language based on paths and processes. *Acta Informatica*, 12(2):451–503, 1979.

[21] W. E. Riddle. An approach to software system behavior modeling. *Computer Languages*, 4:29–47, 1979.

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

[23] 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.

[24] 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.

[25] J. C. Wileden, A. L. Wolf, W. R. Rosenblatt, and P. L. Tarr. Specification level interoperability. Submitted.