# CONSTRAINED EXPRESSION ANALYSIS OF REAL–TIME SYSTEMS

George S. Avrunin[*,1]
Laura K. Dillon[†,2]
Jack C. Wileden[‡,1,3]

COINS Technical Report 89–50
April 1989

*Department of Mathematics and Statistics
University of Massachusetts
Amherst, Massachusetts 01003

†Computer Science Department
University of California
Santa Barbara, California 93106

‡*Software Development Laboratory*
Computer and Information Science Department
University of Massachusetts
Amherst, Massachusetts 01003

---

This work was supported in part by the following grants:
[1]National Science Foundation grant CCR-8806970 and ONR grant N00014-89-J-1064
[2]National Science Foundation grant CCR-8702905
[3]National Science Foundation grant CCR-87-04478 with cooperation from the Defense Advanced Research Projects Agency (ARPA Order No.6104).

# Constrained Expression Analysis of Real-Time Systems

George S. Avrunin[1]
Department of Mathematics and Statistics
University of Massachusetts, Amherst

Laura K. Dillon[2]
Department of Computer Science
University of California, Santa Barbara

Jack C. Wileden[1,3]
Department of Computer and Information Science
University of Massachusetts, Amherst

**Abstract**

The constrained expression formalism and its associated analysis techniques were originally developed for describing and analyzing logical properties of concurrent system behavior. We have recently begun exploring their application to analyzing timing properties. In this paper we describe our initial approach to constrained expression analysis of real-time systems and report the results of a preliminary experiment using this approach. We then discuss the prospects for extending both the constrained expression formalism and our existing prototype toolset to support analysis of real-time concurrent software systems.

[1] Partially supported by NSF grant CCR-8806970 and ONR grant N00014-89-J-1064
[2] Partially supported by NSF grant CCR-8702905
[3] Partially supported by NSF grant CCR-8704478 with cooperation from DARPA (ARPA order 6104).

# 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 intended to be applied. Our *constrained expression* formalism and its associated analysis techniques [6] were developed specifically for analyzing the *logical* aspects of concurrent system behavior. That is, they were tailored to help in uncovering logical flaws or unintended properties in a system's behavior, such as deadlock, process starvation, or synchronization anomalies. We have built a prototype toolset to automate this kind of constrained expression analysis and begun applying it to example concurrent systems [5].

Recently we have started to explore the possibility of applying our constrained expression formalism and analysis techniques to real-time concurrent systems. As an initial step, we carried out an experiment in applying our existing prototype analysis toolset to a simple real-time problem. The results of this experiment were quite encouraging, and we are therefore beginning to design appropriate extensions and modifications to our formalism and tools to automate the kind of analysis that we performed in the experiment. We are also investigating extensions to the formalism that will permit us to describe and analyze real-time systems that employ particular scheduling algorithms or execute on networks of processors or multi-processor machines. In this paper we report on our experiment and our plans for further development of constrained expression analysis of concurrent real-time software systems.

The next section of the paper briefly outlines the constrained expression approach and the third section describes the existing prototype toolset. The fourth describes our initial approach to applying constrained expression analysis to real-time concurrent systems and our experiment in analyzing a simple real-time problem. Finally, we discuss the extensions to the formalism, the analysis techniques and the toolset that will be required to support constrained expression analysis of concurrent real-time systems.

# 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 is given in [10], and a less formal treatment presenting the motivation for many of the features of the formalism appears in [6]. The use of constrained expressions with a variety of development notations is illustrated in [6] and [12].

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 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 [12].

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 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.

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 [9], event expressions [19], and COSY [18]. 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 [6] and [21]. 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., [15]).

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. Ongoing research [5] is directed at determining the practical significance of these potential problems.

After manually applying the constrained expression analysis techniques to a number of small examples with encouraging results (e.g., [3], [6], [7], [21]), we began to construct prototype tools automating various aspects of the analysis. We briefly describe the current version of the prototype toolset before discussing the application of the formalism, analysis techniques and toolset to analyzing concurrent real-time systems.

# 3 The Constrained Expression Tools

The prototype toolset (see Fig. 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 solution found by the integer programming package (when the inequalities are consistent) to produce a string

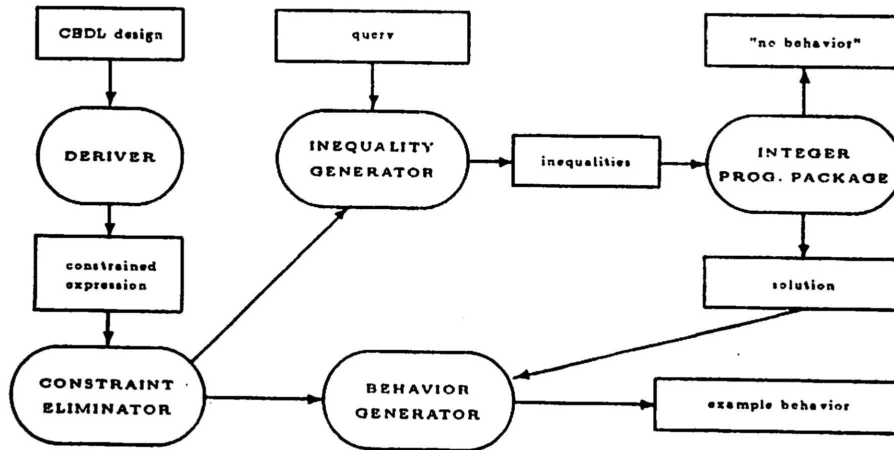


Figure 1: Diagram of Constrained Expression Toolset

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 [5].

The current toolset is intended for use with designs written in the Ada-based design language CEDL (Constrained Expression Design Language) [11]. 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 [20].

The deriver [1] 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 tasks in the system. The deriver also generates all required constraints.

The constraint eliminator [13] takes a subexpression of the system expression 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. This

process facilitates certain aspects of analysis of the constrained expression.

The inequality generator [4] takes a constrained expression representation and generates a system of linear inequalities representing a large part of the semantics of the constrained expression. It 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 we are currently using is a branch-and-bound integer linear programming system [17] written in FORTRAN. 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 [14] 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.

# 4 Applying Constrained Expression Analysis to Real-Time Systems

The constrained expression formalism models computation as a stream of non-overlapping atomic events, with no notion of time. We can, though, introduce time 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. Such an interpretation only makes sense when the events are nonoverlapping, as when the processes in the concurrent system being analyzed are running on a single processor. With this restriction, our prototype constrained expression toolset can be used, for example, to determine bounds on the maximum possible time between two events. In this section, we describe the use of the toolset to obtain information about timing in the CEDL gas station system described in [3] and [21].

This system is a CEDL version of the example used by Helmbold and Luckham [16] to illustrate their run-time monitoring approach to debugging Ada tasking programs. It consists of four tasks, representing two customers, a pump, and an operator. The customers repeatedly prepay, pump gas, and receive change; the operator accepts payment, activates the pump, and gives change to the customers; once activated, the pump dispenses gas and notifies the operator of the charge. The task declarations and bodies are shown in Figures 2, 3, and 4.

To illustrate the kind of timing questions that constrained expression analysis can answer, consider a customer who has just finished pumping (i.e., has just completed a rendezvous at PUMP.FINISH_PUMPING). Constrained expression

```
package COMMON is
  type C_NAME is (cus1,cus2);     -- names for two customers
  type COUNTER is (zero,one,two);  -- enough for two customers
end COMMON;

use COMMON;
task OPERATOR is
  entry PREPAY(CUSTOMER_ID : in C_NAME);
  entry CHARGE;
end OPERATOR;

task PUMP is
  entry ACTIVATE;
  entry START_PUMPING;
  entry FINISH_PUMPING;
end PUMP;

use COMMON;
task CUSTOMER_1 is
  entry CHANGE;
end CUSTOMER_1;

use COMMON;
task CUSTOMER_2 is
  entry CHANGE;
end CUSTOMER_2;
```

Figure 2: Task declarations for the two-customer system

analysis [3, 21] shows that the customer tasks do not starve (i.e., wait indefinitely for a rendezvous) at the CHANGE entries, so the customer will eventually receive its change (i.e., rendezvous with the OPERATOR task at the customer's CHANGE entry). It is then reasonable to ask how long the customer might have to wait for change.

To answer such a question. we have to assign durations to each event in the system. For simplicity, we assume that each event takes one unit of time; although the worst-case waiting time may be different if events have different durations, the method for obtaining it will be the same. Since the two customer tasks are treated symmetrically in the gas station system, there is no loss of generality in assuming that it is the task CUSTOMER_1 that has just completed

```
task body PUMP is
  begin
    loop
      accept ACTIVATE;
      accept START_PUMPING;
      accept FINISH_PUMPING do
        ...     -- compute charge for this transaction
      end FINISH_PUMPING;
      OPERATOR.CHARGE;     -- report charge to operator
    end loop;
  end PUMP;

use COMMON;
task body CUSTOMER_1 is
  begin
    loop
      OPERATOR.PREPAY(cus1);
      PUMP.START_PUMPING;
      PUMP.FINISH_PUMPING;
      accept CHANGE;
    end loop;
  end CUSTOMER_1;

use COMMON;
task body CUSTOMER_2 is
  begin
    loop
      OPERATOR.PREPAY(cus2);
      PUMP.START_PUMPING;
      PUMP.FINISH_PUMPING;
      accept CHANGE;
    end loop;
  end CUSTOMER_2;
```

Figure 3: Bodies of the PUMP and CUSTOMER Tasks

the rendezvous at the entry PUMP.FINISH_PUMPING.

Our procedure is to modify the constrained expression representation of the system to reflect the activity of the system beginning with the time this rendezvous is completed, generate a system of inequalities from this modified

```
use COMMON;
task body OPERATOR is
  CUSTOMERS : COUNTER := zero;
  CURRENT, WAITING : C_NAME;
  begin
    loop
      select
        accept PREPAY(CUSTOMER_ID : in C_NAME) do
          CUSTOMERS := COUNTER'succ(CUSTOMERS);
          if CUSTOMERS = one then      -- if no previous customer
                                       -- is waiting
            CURRENT := CUSTOMER_ID;    -- mark this one as current
            PUMP.ACTIVATE;             -- and activate the pump
           else
            WAITING := CUSTOMER_ID;     -- otherwise, mark this one
                                        -- as next in line
          end if;
        end PREPAY;
      or
        accept CHARGE;
        if CUSTOMERS > one then         -- if another customer is
                                        -- waiting,
          PUMP.ACTIVATE;                -- activate the pump
        end if;
        if CURRENT = cus1 then
          CUSTOMER_1.CHANGE;
         else
          CUSTOMER_2.CHANGE;
        end if;
        CUSTOMERS := COUNTER'pred(CUSTOMERS);
        if CUSTOMERS > zero then     -- if another customer is
          CURRENT := WAITING;        -- waiting, promote that one
                                     -- to be current
        end if;
      end select;
    end loop;
  end OPERATOR;
```

Figure 4: Body of the OPERATOR task

representation, and then use the integer programming package to find a solution to those inequalities. The objective function used for the integer programming package is set to maximize the number of events (or, more generally, the total duration of the events). This maximum gives an upper bound on the amount of time the customer has to wait for change, rather than the precise worst-case time, for reasons discussed below.

In this case, the task expression for CUSTOMER_1 can be reduced to symbols representing only the rendezvous at CUSTOMER_1.CHANGE, since we assume that the customer has just completed a rendezvous at PUMP.FINISH_PUMPING and is therefore waiting for the rendezvous at CUSTOMER_1.CHANGE. Unfortunately, the modifications required for the other task expressions are not as simple. In the experiment described here, we made these modifications manually, after analysis of the possible states of the system at the time that CUSTOMER_1 finishes pumping. We discuss the possible automation of this process in the next section.

We begin by considering the activity of the task CUSTOMER_2. The task expression for CUSTOMER_2 is shown in Figure 6, using symbols whose interpretations are given in Figure 5. This is essentially the task expression produced by the deriver, after some minor simplification. As shown in the figure, several "starvation alternatives" and an "abort" alternative are produced to represent the possibility that the task terminates abnormally. (No normal termination alternative is produced because of the infinite loop in the task body.) The analysis of [3, 21], however, shows that the abnormal termination alternatives never contribute to a behavior of the system. That is, the system has no finite behaviors. For our purposes here, however, we are interested in the number of events occurring between two rendezvous in a (presumably) "infinite" behavior. We can regard an infinite behavior of this system as a limit of a chain of finite prefixes, where the activity of the task CUSTOMER_2 is described in each of the finite prefixes by a string from the language of the expression obtained by deleting the abnormal termination alternatives in the task expression in Figure 6 and each of the finite prefixes satisfies the constraints.

As a first step in transforming the task expression of Figure 6, we therefore eliminate the subexpression containing the abnormal termination alternatives. We then proceed to consider the activity of the task CUSTOMER_2 beginning with the completion of a rendezvous between the tasks CUSTOMER_1 and PUMP at the entry PUMP.FINISH_PUMPING.

Since we are interested in worst-case times, we can ignore the cases where CUSTOMER_2 has begun a rendezvous but not yet completed it. The reason for this is that, if such a rendezvous has begun before the event marking the beginning of the interval being timed (i.e., the completion of the rendezvous at PUMP.FINISH_PUMPING), we can find another behavior in which the rendezvous begins *after* the beginning of the interval. Since the duration of the interval will be greater in the second case, we may ignore the first one. Similarly, the duration of the interval will be greater in the case where CUSTOMER_2 has just completed a rendezvous at OPERATOR.CHANGE than in the case where CUSTOMER_2

| *Symbol* | *Associated event* |
|---|---|
| *def*(V,v) | Variable V is assigned the value v |
| *use*(V,v) | Variable V is assumed to have the value v |
| *beg_loop*(L) | Begin execution of loop L |
| *call*(T,E) | Task T calls entry E |
| *call*(T,E,v) | Task T calls entry E with actual input value v |
| *beg_rend*(T,E) | Begin rendezvous with task T at entry E |
| *beg_rend*(T,E,v) | Begin rendezvous with task T at entry E assuming input value v |
| *end_rend*(T,E) | End rendezvous with task T at entry E |
| *resume*(T,E) | Resume task T after rendezvous at entry E |
| $starve_c$(T,E) | Task T starves on call to entry E |
| $starve_a$(E) | Task starves waiting to accept a call at entry E |
| *kill_rend*(E) | Rendezvous at entry E is aborted |
| *dead_rend*(T,E) | Rendezvous with task T at entry E is assumed to abort |
| *stop*(T) | Execution of task T stops |

**In the symbols used in the task expressions, the task name CUSTOMER_i is abbreviated to Ci, PUMP is abbreviated to P, and OPERATOR is abbreviated to O. Variable and entry names are also abbreviated.**

Figure 5: Event Symbols and Associated Events for CEDL

has not yet started its execution (i.e., is about to call OPERATOR.PREPAY for the first time), since in the first case the OPERATOR has more statements to execute before accepting the next call from CUSTOMER_2 to the entry OPERATOR.PREPAY. Finally, it is not possible for CUSTOMER_2 to have just completed a rendezvous at PUMP.START_PUMPING, since we are assuming that the PUMP has just completed a rendezvous at PUMP.FINISH_PUMPING with CUSTOMER_1, and it is easy to show that successive rendezvous at PUMP.START_PUMPING and PUMP.FINISH_PUMPING must involve the same customer task.

This leaves three cases to consider:

1. CUSTOMER_2 has just completed a rendezvous at OPERATOR.PREPAY,

2. CUSTOMER_2 has just completed a rendezvous at PUMP.FINISH_PUMPING,

3. CUSTOMER_2 has just completed a rendezvous at OPERATOR.CHANGE.

In each of these cases, it is then necessary to determine the subsequent activity of the PUMP and OPERATOR tasks, and to appropriately modify the task expressions of these three tasks and the constraints to reflect that activity. The modified task expression for CUSTOMER_2 corresponding to the first of these cases is shown in Figure 7. Determining the activities of the PUMP and OPERATOR tasks, given those of the two customer tasks, is straightforward.

We now have three constrained expression representations, corresponding to the three cases. After applying the constraint eliminator to the individual task expressions, we use the inequality generator to produce systems of inequalities.

$$beg_loop(\mathrm{C2})\Big(call(\mathrm{C2, O.prepay, cus2})resume(\mathrm{C2, O.prepay})call(\mathrm{C2, P.start})$$

$$resume(\mathrm{C2, P.start})call(\mathrm{C2, P.finish})resume(\mathrm{C2, P.finish})$$

$$beg_rend(\mathrm{O, C2.change})end_rend(\mathrm{O, C2.change})\Big)^{*}$$

$$\Big(starve_c(\mathrm{C2, O.prepay})stop(\mathrm{C2})$$

$$\vee call(\mathrm{C2, O.prepay, cus2})dead_rend(\mathrm{C2, O.prepay})stop(\mathrm{C2})$$

$$\vee call(\mathrm{C2, O.prepay, cus2})resume(\mathrm{C2, O.prepay})starve_c(\mathrm{C2, P.start})$$

$$stop(\mathrm{C2})$$

$$\vee call(\mathrm{C2, O.prepay, cus2})resume(\mathrm{C2, O.prepay})call(\mathrm{C2, P.start})$$

$$resume(\mathrm{C2, P.start})starve_c(\mathrm{C2, P.finish})stop(\mathrm{C2})$$

$$\vee call(\mathrm{C2, O.prepay, cus2})resume(\mathrm{C2, O.prepay})call(\mathrm{C2, P.start})$$

$$resume(\mathrm{C2, P.start})call(\mathrm{C2, P.finish})resume(\mathrm{C2, P.finish})$$

$$starve_a(\mathrm{C2.change})stop(\mathrm{C2})\Big)$$

Figure 6: Task Expression for CUSTOMER_2

For each of these three systems, we use the integer programming package to find the solution with the maximum number of events (or, if the events have different durations, the maximum total duration). The maximum of the number of events in the three solutions is the upper bound we seek.

There are two reasons why this upper bound may not be sharp. First, the inequalities produced by our tools do not reflect the full semantics of constrained expressions, and the solution found by the integer programming package may not correspond to a behavior of the system that is actually possible. In these cases, we manually generate additional inequalities that eliminate such spurious solutions and improve the bounds. Second, we do not yet know how to "stop" the other tasks in the system exactly at the occurrence of the event marking the end of the interval being timed. Thus, in the gas station example, the other

$$call(\mathrm{C2}, \mathrm{P.start})\, resume(\mathrm{C2}, \mathrm{P.start})\, call(\mathrm{C2}, \mathrm{P.finish})$$

$$resume(\mathrm{C2}, \mathrm{P.finish})\, beg_rend(\mathrm{O}, \mathrm{C2.change})\, end_rend(\mathrm{O}, \mathrm{C2.change})$$

$$\Big(call(\mathrm{C2}, \mathrm{O.prepay}, \mathrm{cus2})\, resume(\mathrm{C2}, \mathrm{O.prepay})\, call(\mathrm{C2}, \mathrm{P.start})$$

$$resume(\mathrm{C2}, \mathrm{P.start})\, call(\mathrm{C2}, \mathrm{P.finish})\, resume(\mathrm{C2}, \mathrm{P.finish})$$

$$beg_rend(\mathrm{O}, \mathrm{C2.change})\, end_rend(\mathrm{O}, \mathrm{C2.change})\Big)^*$$

Figure 7: Task Expression for CUSTOMER_2 in Case 1

tasks in the system may continue to run after CUSTOMER_1 receives change. This also makes it impossible, in general, to use these methods to obtain valid lower bounds for the time between two events. Ongoing research addressing these problems is discussed in the next section.

For the system analyzed here, the upper bound occurs in the case where CUSTOMER_2 is assumed to have just completed a rendezvous at OPERATOR.PREPAY when timing starts. The solution found by the integer programming package does correspond to a real system behavior, with the following sequence of activities:

> CUSTOMER_2 prepays; OPERATOR activates the PUMP; CUSTOMER_1 prepays; CUSTOMER_1 starts pumping; CUSTOMER_1 finishes pumping [timing begins]; PUMP reports the charge; OPERATOR activates the PUMP; CUSTOMER_2 starts pumping; CUSTOMER_2 finishes pumping; OPERATOR gives change to CUSTOMER_2; CUSTOMER_2 prepays; PUMP reports the charge; OPERATOR activates the PUMP; CUSTOMER_2 starts pumping; CUSTOMER_2 finishes pumping; OPERATOR gives change to CUSTOMER_1 [timing ends]; PUMP reports the charge; OPERATOR gives change to CUSTOMER_2.

Events taking place before the start of timing are eliminated by the modification of the task expressions and do not contribute to the bounds, but events occurring after the end of the timed interval are included in the calculation of the bound. Thus, the bound we obtain is inflated by the inclusion of the events involved in the last report of charges by the PUMP and the return of change to CUSTOMER_2. (Of course, CUSTOMER_2 can continue to prepay, pump gas, and receive change repeatedly after the end of the timed interval. We impose a limit on the number of times CUSTOMER_2 prepays in order to find solutions to the

integer programming problems.) The reader will also have noticed that the customers receive the wrong change in this behavior. This version of the gas station represents one stage in an iterative process of development used by Helmbold and Luckham [16] to illustrate their debugging method; at this stage, design flaws in earlier versions that led to possible starvation have been corrected but the problem with the distribution of change remains.

# 5 Current Research

The experiment reported in this paper demonstrates that it is possible to apply constrained expression analysis techniques and some components of our prototype toolset to obtain useful information about real-time properties of concurrent systems. At the same time, it indicates several research directions that should be pursued in order to increase the applicability of constrained expression analysis to real-time systems. We are currently seeking to develop better constrained expression formulations of those aspects of concurrent system behavior that are relevant to real-time properties. We are also exploring extensions to the constrained expression approach to behavior description that will permit us to analyze real-time properties of additional classes of concurrent systems. In particular, we are considering ways to express specific scheduling policies in a constrained expression behavior representation and techniques for representing the execution of multi-processor systems, in which event occurrences on different processors may be regarded as overlapping in time. Finally, we are working on improving automated support for constrained expression analysis of real-time concurrent systems. Our ongoing research in each of these areas is briefly described below.

In the experiment reported in this paper, we used a simple, expedient approach for associating real-time properties with pre-existing constrained expression representations of the gas station system's behavior. That approach resulted in a somewhat complicated analysis procedure, parts of which may be difficult to automate. Alternative approaches to capturing real-time properties in a constrained expression could improve the effectiveness and efficiency of our analyses. We are therefore investigating improved constrained expression formulations of the aspects of concurrent system behavior relevant to real-time properties. For example, we are exploring various alternatives for representing the passage of time by "tick events", rather than time durations associated with event symbols. One possibility is to have global clock ticks emitted by a single "clock task expression", with constraints introduced to ensure that event symbols from the same task expression are separated by the appropriate number of clock ticks. Another is to have local clock ticks emitted as part of each task expression, with constraints to enforce the synchronization of corresponding ticks from different tasks. The different alternatives may have implications for the types of analysis that can be performed. We are also investigating alternate ways

of modifying the constrained expression representation of a system to reflect the timing property of interest. For example, we are considering the possibility of inserting a "start timing" event symbol and a "stop timing" event symbol into a constrained expression, then using appropriate constraints to filter out all behaviors except those that are delimited by this pair of event symbols. Under such an approach, the analysis performed in our experiment would reduce to finding the longest such behavior. The interaction between this approach and our current treatment of constraints has many subtleties, however, and its potential benefits have yet to be investigated in detail. We are currently trying to understand how these different formulations affect constrained expressions analysis.

When analyzing logical properties of concurrent software systems it is not appropriate to assume any particular scheduling policy; tasks can run at their own unpredictable rates, and so all possible interactions must be considered equally likely to occur. Similarly, analysis of logical properties of concurrent systems seldom requires explicit attention to the possibility that two events may actually overlap in time; modeling concurrency by arbitrary interleaving of events is generally sufficient to uncover any possible logical flaws. Hence our constrained expression behavior descriptions have never expressed scheduling policies and have only represented overlapping event executions by way of interleaved, atomic initiation and termination events. In analyzing real-time properties of concurrent systems, however, we would like to account for the effects of a specific scheduling policy or the degree of multiprocessing available. Hence, we are exploring extensions to the constrained expression approach to behavior description that will capture these aspects of a system.

Our approach to describing scheduling policies is to introduce constraints that enforce a particular ordering on events based upon the task to whose behavior they belong. For example, with our student Susan Avery, we are currently investigating a constraint that enforces round robin scheduling by forcing successive events in a behavior to come from successive tasks in the round robin sequence. A more complicated constraint to enforce a simple static priority scheduling policy is also being explored. This approach to describing scheduling requires the introduction of new kinds of event symbols, with corresponding modifications to the existing set of constraints and the translation schemes that generate constrained expressions from other descriptions (e.g., CEDL). Detailed investigation of the ramifications of these extensions, their applicability to additional scheduling policies and their impact on the analysis techniques remains to be done.

We are exploring several approaches to constrained expression representation of the event overlap that can occur in multiprocessing systems. One of these is based on the "tick events" discussed above. Under the global clock formulation, simultaneity would be expressed by using a single clock tick to represent passage of time for all tasks. Under the local clock formulation, simultaneity would be expressed by constraints that ensure tasks have emitted the same number of

local clock ticks at the points where they synchronize. Another approach to representing simultaneous activity in multiprocessing systems relies on dividing all events into equal-duration "quantum" units, then using a constraint like that used for round robin scheduling to enforce a "perfect shuffle" of quantum events. Each complete cycle through the round robin would then correspond to the passage of the same time quantum on each processor. Our investigation of both these approaches, their compatibility with our current behavior representation and analysis techniques, and their interaction with the approaches to representing scheduling policies is still in the preliminary stages.

Our approaches to automating real-time analysis involve both increasing the automation of the technique used in our example and developing automated support for other, extended techniques. Regarding the former, it seems unlikely that the type of reasoning that led to the identification of the three cases representing the subsequent activity of the task CUSTOMER_2 in the example above will be readily automated. Given the last event that each task engages in, however, we believe that we may be able to automate the kind of "stripping" of prefixes used in our experiment by employing a technique similar to Brzozowski's regular expression derivative technique [8]. Such an approach has already been applied to constrained expressions in one of the early prototype implementations of the behavior generator tool [2]. Automating the approach to enforcing stopping that was used in our experiment is more complicated. We may, however, be able to use a sort of "successive approximation", in which inequalities imposing successively shorter execution lengths on tasks are successively introduced until an impact on behavior is noted. As our experiment demonstrated, once these two aspects of modifying the constrained expression have been completed, the remainder of the analysis can be carried out using our current prototype toolset. The probable difficulty of automating these two steps, however, has inspired the various alternative approaches to capturing real-time properties in a constrained expression that were described earlier in this section. The prospects for automating analysis of behaviors described using these alternative constrained expression formulations are currently being investigated.

# 6 Conclusions

The constrained expression formalism and its associated analysis techniques were originally developed for describing and analyzing logical properties of concurrent system behavior. Our previous research has demonstrated that the constrained expression approach is applicable to descriptions written in a wide range of notations, covering asynchronous message-passing as well as synchronous communication mechanisms (as in CEDL), and including such notations as Petri nets and CSP [12]. We have also recently produced a prototype toolset that automates much of our constrained expression analysis, and experimentally demonstrated that its performance is adequate to support practical automated

analysis of interesting concurrent systems [5]. In this paper, we have described a preliminary experiment in using constrained expression analysis techniques to analyze timing properties of concurrent software systems. With only minor modifications to standard constrained expression analysis techniques, we were able to obtain upper bounds on the maximum total elapsed time between designated event occurrences.

Our results are encouraging for several reasons. First, they demonstrate a potentially useful approach to carrying out the specific kind of real-time analysis on the particular class of concurrent systems that was the subject of our experiment. More importantly, our current research suggests that we will be able to generalize both the kind of analysis and the class of real-time systems to which the approach can be applied. Furthermore, the performance data from experiments with our existing toolset plus our current research on automating the real-time analysis techniques suggest that it may be possible to provide much of the constrained expression approach to analysis of concurrent real-time systems via practical automated tools. Finally, given our previous demonstration of the broad applicability of the constrained expression approach, the benefits of this work will not be restricted to a single real-time system design notation or programming language, but should be applicable with a wide range of system description techniques.

# References


[1] S. Avery. A tool for producing constrained expression representations of CEDL designs. In preparation.

[2] S. Avery. Development of a behavior generator for constrained expressions. Software Development Laboratory Memo 84-2, Department of Computer and Information Science, University of Massachusetts, Amherst, June 1984.

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

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

[5] G. S. Avrunin, L. K. Dillon, and J. C. Wileden. Experiments in automated analysis of concurrent software systems. Submitted.

[6] 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.

[7] 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.

[8] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4):481–494, Oct. 1964.

[9] 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.

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

[11] 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.

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

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

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

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

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

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

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

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

[20] 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.

[21] 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.