AUTOMATED FORMAL ANALYSIS METHODS FOR CONCURRENT AND REAL-TIME SOFTWARE

James C. Corbett

COINS Technical Report 92–48
September 1992

Computer Science Department
University of Massachusetts
Amherst, MA 01003

Submitted to the Graduate School of the
University of Massachusetts in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 1992

Computer Science

This work was supported by the National Science Foundation under grants CCR-8806970 and CCR-9106645 and the Office of Naval Research under grant N00014-89-J-1064.

Acknowledgements

Most of all, I would like to thank my advisors George Avrunin and Jack Wileden for the support and instruction they have given me during my graduate career. They encouraged me when I needed it, showed me how to express my ideas clearly, and gave me a sense of what is important in research. I have learned a lot from both of them.

I would also like to thank the other members of my committee, David Mix-Barrington and Krithi Ramamritham, for a careful reading of this document and helpful suggestions for its improvement. I would especially like to thank Dave, who was my advisor for the first two years of my graduate study and first inspired me with an enthusiasm for research.

Finally, I would like to thank Allyn Polk, who introduced me to constrained expressions, and Ugo Buy, who spent a lot of time bringing me up to speed on the project.

Abstract

Automated Formal Analysis Methods for Concurrent and Real-Time Software

September 1992

James C. Corbett, B.S., Rensselaer Polytechnic Institute

M.S., University of Massachusetts

Ph.D., University of Massachusetts

Directed by: Professor George S. Avrunin
Professor Jack C. Wileden

As the use of concurrent and concurrent real-time software systems in safety-critical applications becomes widespread, the verification of their correctness has become an important concern. Unfortunately, analysis of these systems has been stymied by the explosive number of states they possess. The *constrained expression* approach, which uses an inequality-based technique to avoid the enumeration of these states, showed promise for analyzing large systems, but was incapable of verifying many important properties of interest to designers. For example, properties involving the order of the events in a concurrent system (e.g., mutual exclusion) could not be verified since the inequalities did not capture this information, nor could the technique verify liveness properties, since these require reasoning about infinite executions. I have developed extensions to this inequality-based technique that allow the verification of these more complex properties. In addition, I have completely automated an earlier extension of this technique for deriving bounds in concurrent real-time systems run on a uniprocessor and I have extended this technique to the maximally-parallel multiprocessor setting. Most importantly, I have demonstrated the feasibility of these extensions by implementing them in an automated tool and using this tool to analyze several sample systems.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

INTRODUCTION

As the use of concurrent and concurrent real-time software systems in safety-critical applications becomes widespread, the verification of their correctness has become an important concern. Unfortunately, assessing the correctness of these programs is difficult. Establishing the correctness of sequential programs, even in the weakest sense of proving conformance to specifications, is hard enough without the complexity added by concurrency. Manual reasoning about the many possible orderings of events that can occur in concurrent systems is error-prone and existing techniques overwhelm the analyst with detail when applied to systems of realistic size.

Therefore, to build larger and more reliable concurrent systems, future software developers will need automated tools that facilitate this reasoning. Such tools should aid the software developer in establishing that an implementation of a piece of software meets its specification. The specification of a concurrent program will contain different types of requirements. Some will specify the required input/output behavior of the program. Others will specify concurrency properties, such as freedom from deadlock. Still others might specify timing requirements.

The work presented here focuses on the validation of concurrency and timing properties of concurrent software by automated analysis tools. My goal is to build practical tools that can help software developers establish whether a concurrent system has a certain concurrency or timing property. Practicality requires that these tools have reasonable running times on realistically-sized programs and be largely automatic (i.e., not require extensive user assistance).

The *constrained expression* approach [8, 10] to the analysis of concurrent systems shows promise as a practical analysis technique. Unlike most, this analysis technique does not require the enumeration of a potentially explosive number of system states, and, unlike logic-based proof techniques, it can be feasibly automated with current technology. My thesis significantly extends this approach, mainly in terms of the range of properties that these systems can be shown to possess.

The rest of this chapter gives an overview of the concurrency and timing properties of interest to system designers, together with a high-level description of the contributions that this dissertation makes toward the practical verification of these properties. Chapter 2 outlines general methods for validating software and discusses closely related work, while Chapter 3 gives a detailed review of the constrained expression approach, which this thesis extends. The next four chapters compose the body of the dissertation. Chapter 4 describes two techniques that allow larger concurrent systems to be analyzed. Chapter 5 describes several techniques for verifying a range of more complex concurrency properties. Chapter 6 describes techniques for deriving bounds on the time between events in concurrent real-time systems. In Chapter 7, I describe the implementation of a prototype tool automating many of the techniques developed in Chapters 4, 5 and 6 and present the results of experiments in which this tool is applied to several sample systems. Finally, Chapter 8 concludes by discussing the limitations and future extensions of the work.

1.1 Properties of Interest

Before delving into analysis techniques for concurrent and real-time software, I first review the properties of such software that these techniques seek to establish. I divide these properties into concurrency properties and timing properties.

Concurrency properties involve the possible sequences of interactions between the tasks, or processes, composing the concurrent program. Each property is either desirable or undesirable. Further, each property is classified as a *safety property*

```
task body one is
begin
  loop
    two.A;
    -- Critical
    -- Section
  end loop;
end one;
```

```
task body two is
got_b:boolean:=false;
begin
  while not got_b loop
    select
      accept A;
      -- Critical
      -- Section
    or
      accept B;
      got_b := true;
    end select;
  end loop;
end two;
```

```
task body three is
begin
  two.B;
  -- Critical
  -- Section
end three;
```

Figure 1.1. Example to Illustrate Properties of Interest

or a *liveness property.* Informally, safety properties are statements of the form "Something bad will never happen" whereas liveness properties are statements of the form "Something good will eventually happen." An execution of the concurrent program up to some point (where the "bad thing" has happened) can show the violation of a safety property, but only a full execution can show the violation of a liveness property. Some concurrency properties are quite specific; these include deadlock, starvation, critical races, and mutual exclusion. More general properties concern the reachable states of the system or possible patterns of events. The following paragraphs will discuss each of these properties in turn, using the small concurrent system in Figure 1.1 as an example. The system is coded in an Ada-like specification language used by the constrained expression tools. For those not familiar with Ada, statements such as `two.A` and `accept A` are matching synchronous communication statements for entry `A` of task `two`. A task reaching one of these statements must wait for the other task to reach a matching statement before both can then proceed. The `select` statement allows a nondeterministic choice between the communication statements it contains.

In the context of communicating tasks, *deadlock* is the situation in which one or more tasks become permanently blocked waiting for communication that cannot

occur. Typically, several tasks are involved in a deadlock, each task waiting to communicate with another in some way that is not currently possible. A single task can be said to deadlock if it waits for communication with a task that will never be willing to engage in that communication. In the example of Figure 1.1, once the B communication has taken place, task **one** is deadlocked waiting for an A communication that is no longer possible since task **two** has terminated. Freedom from deadlock is generally a desirable safety property.

Related to deadlock is the concept of *starvation.* A task is said to starve waiting for a communication in an infinite execution if it never makes progress. In this case, it is not that the communication cannot occur, it is simply the case that the communication never does occur because of the way communication partners are selected whenever there is a choice. In the example of Figure 1.1, it is possible for task **three** to starve waiting for the B communication if task **two** chooses to engage in the A communication forever. Absence of starvation is usually a desirable liveness property.

A *critical race* is a situation in which either of two statements from different tasks can execute before the other, but the behavior of the concurrent program will be affected by the outcome of this "race". Sometimes critical races are a natural part of a system, as would be the case if the two statements were allocations of a resource to competing customers. They are frequently the source of errors, however, when the programmer assumes that one specific task always wins the race. For example, one task may pass data to another by writing it to a global variable that the other then reads. If nothing prevents the read from executing before the write, then the critical race between these two statements could result in incorrect data being passed. Even without global data, critical races can be present in the communication protocol used by tasks in a concurrent program when two tasks race to communicate with a third task. In the example of Figure 1.1, tasks **one** and **three** race to communicate with task **two**. Absence of unintended critical races is a desirable safety property.

Mutual exclusion requires that two tasks never execute certain sections of their code, called *critical sections*, at the same time. In the example of Figure 1.1, mutual exclusion is not enforced between the critical sections of tasks `one` and `three`, but is enforced between the critical sections of tasks `two` and `three`. Usually used to guarantee the integrity of a shared resource, mutual exclusion is a desirable safety property.

More general properties, subsuming all of the above, can be expressed in terms of the possible states or sequences of events of the system. Detection of properties like deadlock or violation of mutual exclusion are specific instances of the *reachability problem*, which asks whether the system can reach a state having some property (e.g., all tasks are blocked, two tasks are both in their critical sections, etc.). Properties involving sequences of events are usually expressed in *temporal logic* [62], but can also be expressed using regular expressions or automata. For example, a liveness property, such as absence of starvation, might be expressed by a formula that requires that a request for a resource eventually be followed by the granting of that resource to the requesting task.

The second group of properties to review are those relating to timing requirements. Interest in formal verification of real-time systems is more recent than interest in verification of concurrent systems and perhaps as a result there are fewer standardly assessed timing properties. The most common timing information gathered from analysis is upper and lower bounds on the time that can elapse between events in a program [26]. This information can be used to determine bounds on the execution time of program segments, which in turn can be used to guarantee that tasks complete before their deadlines. More complex timing requirements can be expressed in a real-time logic, which has predicates, variables, and quantifiers over relative or absolute time. A typical requirement is that a particular action be performed once in every period of time p.

1.2 Contributions of this Dissertation

This section summarizes the contributions of this dissertation. These are primarily extensions to the constrained expression analysis technique that allow the verification of many important properties that the original technique was unable to address. Like the work with the original technique described in [8], my extensions focus on concurrent systems that use synchronous communication, but can be adjusted for use on systems with an asynchronous communication mechanism (though I have *not yet* experimented with the techniques on such systems).

Since most concurrent systems can be modeled as finite state machines, the main obstacle to verifying the correctness of these systems is not undecidability but intractability. By enumerating the possible states of a concurrent system, all of the properties in Section 1.1 are decidable. This enumeration is usually intractable in practice, however, due to a state explosion: the number of system states is typically exponential in the number of tasks in the system.

A variety of techniques have been proposed to overcome this state explosion (many of these are outlined in Section 2.2). Among them is the constrained expression approach, described in Chapter 3, which has been successfully applied to a variety of concurrent systems, some having as many as 10^{47} reachable states [7, 8]. By generating a system of linear inequalities that represent necessary conditions for the existence of an execution with certain properties, the approach avoids the enumeration of the system states altogether.

The approach described in [8] was limited, however, to the verification of properties expressible in terms of the total numbers of various events occurring in a finite execution. Although freedom from deadlock can be so expressed, many important properties of interest cannot be. Since the inequalities do not capture the order of the events specified, properties involving the order of events, such as mutual exclusion, could not be directly verified. Also, since a violation of a liveness property is an entire execution, liveness properties could not be verified for concurrent systems

having infinite executions, as most (conceptually) do. Critical races, perhaps the most common source of errors in concurrent software, could not be detected by the technique since the presence of a race must be shown by displaying two executions in which different tasks win the race.

As for timing analysis, the approach had been extended to derive upper and lower bounds on the time that can elapse between the occurrence of two events in a concurrent real-time program [11], but the bounds derived by this technique were good only in a uniprocessor setting. Also, the analysis required significant assistance from the analyst and was therefore not practical on systems of significant size.

My contributions extend the original constrained expression analysis technique in several ways, removing many of its limitations and extending the size of the systems that can be analyzed. These contributions can be grouped into three categories: extensions allowing larger systems to be analyzed, extensions allowing new concurrency properties to be verified, and extensions of the timing analysis. I discuss each of these in turn.

I have developed two techniques that allow larger systems to be analyzed. The first technique allows efficient reasoning about systems with many identical tasks. The second technique allows efficient reasoning about tasks with a certain common type of variable (a counter) having a large range.

Many concurrent systems have a large number of identical tasks. Researchers using reachability analysis have already noticed that the size of the state space of such a system can be significantly reduced using this symmetry. I have developed a way to represent an essentially arbitrary number of identical tasks using only one task in the context of a constrained expression analysis. I say "arbitrary" because only some coefficients in the inequality system change as the number of copies of a task increases—the size of the inequality system and, it appears, the time to solve the system, remain constant (the limits here are imposed by numerical stability in solving the inequality system).

I have also developed a way to represent the value of a program variable that represents a counter (i.e., a variable that is only incremented and decremented) using an integer linear programming variable. Typically, the size of a task's representation (as an automaton or regular expression) increases with the number of values its program variables can take on. With this technique, the size is independent of the ranges of the task's counter variables. I have conducted an experiment with these two techniques for analyzing larger systems on a resource allocator problem that has a deadlock only if there are at least n customer tasks, where n can be very large. The toolset correctly determined if deadlock was possible in versions with 500 and 1000 identical customer tasks. Furthermore, the analysis times for these two versions were very small, and exactly the same.

The second set of extensions to the original analysis technique allows new concurrency properties to be verified. Three new types of properties can now be verified: safety properties involving the order of events, liveness properties, and critical races. I briefly describe each of these extensions.

I have extended the constrained expression approach to reason about executions in which events occur in a specific order. The original analysis technique can verify properties that can be expressed in terms of the total numbers of different events in an execution, but cannot verify properties involving only the order of those events. For example, it can determine whether there exists an execution in which two specific events occur, but it cannot determine whether there exists an execution in which a particular one of these events occurs before the other. Since properties such as mutual exclusion involve the order of events (e.g., user 2 may not gain access to a shared resource after user 1 has gained access, but before user 1 has relinquished access), they cannot be addressed directly by the original technique. By dividing the execution into segments, generating an inequality system for each segment, and then connecting these systems together, I can determine whether there exist executions in which certain events occur in a specific order.

The constrained expression formalism uses finite strings to represent the behavior of a concurrent system, thus the original analysis technique, based on this formalism, could not reason about infinite executions. As a result, the technique could not address liveness properties since an execution violating such a property may be infinite. By using *perpetual symbols*, symbols that represent the infinitely repeated occurrence of an event, I have developed a way to represent an infinite execution approximately using a finite string, allowing the technique to verify liveness properties.

Another extension to the original technique can aid in the detection of critical races. The new technique can determine whether two tasks can race to communicate with a third task. While the original technique generated an inequality system representing necessary conditions for the existence of a single execution, this new technique generates two inequality systems, connected appropriately, as necessary conditions for the existence of two closely related executions in which different tasks win the race.

The third set of extensions to the original technique improve its application for timing analysis of concurrent real-time systems. The two major extensions here are automating a technique for deriving bounds on concurrent programs run in a uniprocessor setting, and extending this technique to derive bounds when the program is run in a maximally-parallel multiprocessor setting.

While the constrained expression analysis technique has primarily been applied to analyze logical properties of concurrent programs, it can also be used to analyze certain timing properties of concurrent real-time systems. Previously, the constrained expression formalism and analysis technique had been extended to derive upper and lower bounds on the execution time between two events of a concurrent program run in a uniprocessor setting. Much of the analysis had to be done manually by the analyst, however, so the approach was limited to use on systems small enough for this to be manageable. By using additional integer programming variables and a

graph marking algorithm, I have completely automated this analysis and improved the quality of the bounds obtained.

I have also extended this work to a multiprocessor model. As a first step towards analysis in a general multiprocessor setting, I have developed a technique for the simplest case where each task has its own processor (i.e., *maximal parallelism*). The technique involves generating another set of equations that find the parallel execution time of an execution by finding the critical path through that execution. The equations from the uniprocessor analysis technique, which find the execution itself, are then combined with these new equations so that a bound on the parallel execution time of any execution is found by obtaining the optimal integral solution to a linear system.

Believing that the value of formal methods for software engineering lies in their practical application, I have built a prototype tool implementing these techniques as part of the constrained expression toolset [8] and empirically validated the feasibility of these techniques as the basis for automated analysis tools. Further experiments in which the system size is scaled up, like those done with the original technique in [8], are needed to assess fully the practicality of these techniques. Nevertheless, the great success of the constrained expression approach in analyzing a variety of types and sizes of concurrent systems, along with my preliminary experience with these new techniques, leaves me optimistic that practical analysis tools that can automatically verify complex properties of large software systems are not long in coming.

CHAPTER 2

RELATED WORK

This chapter summarizes related work. Section 2.1 presents a general overview of approaches for assessing the correctness of software, while Section 2.2 focuses on techniques for concurrency analysis, the work most closely related to my own.

2.1 General Approaches

I divide techniques for assessing the correctness of software into three categories: testing, analysis, and verification. Testing involves repeatedly running the program under varying circumstances in the hope that, if the program has an anomalous property, then some behavior of the program in which this is manifest will be observed. Analysis determines whether a program has a certain property by reasoning about its possible executions. Verification involves constructing a proof in a formal system that the system has a certain property. The boundaries between these three categories are fuzzy and the terminology, reflecting a lack of consensus in the field, is not completely standard. Another common categorization divides the approaches into two groups: *dynamic* techniques that involve running the program (i.e., testing), and *static* techniques that do not (i.e., verification, most analysis techniques). I will use the former categories and briefly summarize each in the following sections. This discussion is not an exhaustive survey of these areas; it is intended only to describe some representative work in each category.

2.1.1 Testing

Testing is the most commonly used approach to validating software. The program is run with a variety of inputs and its outputs and behavior are checked for conformance with the specifications. Dijkstra's now famous saying asserts that testing can

only prove the presence of errors, not their absence. In fact, exhaustive testing (testing all possible inputs) can prove the absence of errors, but is too often intractable due to the huge number of possible inputs. Testing focuses on selecting input data that will expose faults. Methods for selecting data could depend on the problem domain (e.g., a statistical selection of typical input), the function the program computes (e.g., test boundary values), or the structure of the program (e.g., exercise all statements, branches, paths, etc.). A survey of testing techniques is given in [1] and more recent results in [46, 59].

For concurrent programs, exhaustive testing would have to consider all possible schedules for the actions of the tasks. Since the number of schedules is usually exponential in the number of tasks, this is almost always intractable. Instead, concurrent programs are run over and over in the hopes that any anomalies will surface. The work on concurrent debugging has focussed on helping the tester to determine when an anomaly has occurred and aiding in the reproduction of that anomaly. Helmbold and Luckham [38] have a run time monitoring system for Ada tasking programs that can detect communications deadlock and show the tester at what statement each task is blocked. Bates and Wileden [13] use an approach called *behavioral abstraction* to reduce the amount of information recorded in a trace of a concurrent system by detecting and recording higher level events. The Instant Replay system of Leblanc and Mellor-Crummey [49] models all process interaction as a shared resource with a version number. This allows the behavior of a concurrent system to be efficiently recorded during an execution and reproduced if desired. Tai and Obaid [70] have a procedure to convert an Ada tasking program into an equivalent one which takes as additional input the sequence of rendezvous recorded in a trace of the program's execution. This new program forces that sequence of rendezvous to occur again, allowing a human debugger to reproduce and examine the execution sequence leading to an anomaly.

Real-time systems are most often validated using testing due to the lack of good verification and analysis techniques. Typically, the program is run many times and the maximum observed execution time is used as an approximation of the actual longest execution time. Razouk and Gorlick [66] have a real-time interval logic that allows complex timing properties to be expressed. Given a specific trace of a real-time system, it is possible to automatically check whether the execution satisfies the real-time specification expressed in the logic. An example specification might require that interrupts be serviced within a specific time interval when they are enabled.

2.1.2 Analysis

Analysis of programs can range from manual inspections or walkthroughs to the use of automated tools that detect errors or anomalies from program text. Analysis focuses on detecting faults or possible faults in a program by reasoning about certain elements of its possible executions. Unlike testing, the program is not actually run, and unlike what I am calling verification, only certain properties of the program are established (i.e., not conformance to a complete specification).

Several powerful techniques have been developed for automatically deriving useful information about a program without running it. Dataflow analysis [57] was developed for optimization of compiled code but can also detect anomalies involving the definition and use of variables. For example, some of these techniques can detect the erroneous possibility that a variable value is used before it has been defined, or the anomalous possibility that a variable value is assigned but never used. The program dependence graph [31] incorporates information on both data and control dependence between statements. It has been used for compiler optimization to determine when pieces of code can be moved within a program in a way that preserves the meaning of the program. Symbolic execution [57] of a program to a specific point generates a boolean expression over the symbolic input values, called the *path condition*, that can be used for documentation, error detection, or test data generation. To detect

errors with symbolic execution, the programmer would add assertions to points in the program expressing what he/she believes to be necessary conditions on the variable values upon reaching these points. When the symbolic executer reaches one of these points, it verifies that the negation of the assertion at that point is inconsistent with the path condition at that point, thus verifying that the assertion will always hold there.

Analysis of concurrent systems has focussed on a variety of properties, such as those mentioned in the introduction. These properties can usually be determined by examining the full reachability graph of the concurrent system. The full reachability graph contains all possible global states of the system, where a global state consists of a tuple of local states of all the component processes. The local state of a process includes both its program counter and the values of its variables. There is an edge from one global state to another in the reachability graph if there is some single action the system can take in the first state that leaves it in the second state. An action may change the states of more than one process (e.g., a synchronous communication). The full reachability graph will not be finite if one of the processes has a variable with an infinite domain and thus is not finite state. Also, the full reachability graph contains all possible interleavings of the actions of the component processes and as such its size is almost always exponential in the number of processes. For these reasons, almost all concurrency analysis techniques assume that their component processes are finite and most do not construct the full reachability graph (since this is as intractable as exhaustive testing) but rather one with only a representative set of interleavings sufficient to verify the properties of interest. Since this work is the most closely related to my own, it is outlined in depth in Section 2.2.

Analysis of real-time systems has essentially consisted of bounding everything and adding up the times. Representative work has been done by Shaw [69]. Each atomic statement is assigned an upper and lower bound on its execution time based on the hardware and machine code used to realize it. The bounds for a composite

statement can be derived from the the semantics of the statement and the bounds on the statements of which it is composed. All loops, communication delays, frequencies of interrupts, etc. must be bounded as well. Often, the upper bounds derived could be very much larger than the actual upper bounds given the environment in which the real-time system will operate. Puschner and Koza [64] have introduced language constructs that allow the programmer to build into the program assumptions about the environment and input values that allow the timing bounds to be tightened.

2.1.3 Verification

One approach to validating the correctness of programs is to prove in formal logic that the implementation of the program satisfies its specification. For sequential programs, the computation is usually considered as a function that takes input values and produces output values. There are two kinds of correctness one can prove for such programs. A proof of *partial correctness* of a program guarantees that *if* the program halts it will produce the right answer, but does not guarantee that the program will halt. A proof of *total correctness* guarantees both eventual termination of the program and partial correctness.

Our current notion of how to construct these proofs originated with Floyd [33] and similar proof systems for sequential programs have been proposed by Hoare [40], Dijkstra [28], Gries [37], and Backhouse [12], among others. The notation $\{P\}S\{Q\}$ denotes that if execution of statement S is started in a program state in which predicate P is true, then the predicate Q will be true in the state after S executes. For example, $\{x \geq 0\}x := x + 1\{x \geq 1\}$ is true. Proofs involve finding the *weakest precondition* on the input values of a program P that will guarantee that if the program terminates it will be in a state that satisfies some predicate expressing correctness. For example, the weakest precondition of the trivial program $x := x + 1$ to guarantee $x \geq 1$ is $x \geq 0$. All of the above proof systems have means to derive the weakest precondition of a primitive statement in isolation, as well as rules for

deriving the weakest precondition of structured compositions of other statements. If the specification of acceptable inputs to the program logically implies the weakest precondition under which it will execute correctly, then we may conclude that the program is partially correct. Proofs of total correctness in sequential programs involve proving that progress is made in each loop toward its completion.

Several approaches to verifying concurrent systems have been proposed. Owicki and Gries [61] extended Hoare's proof system to reason about concurrent programs by introducing the idea of *non-interfering* proofs. The problem with Hoare's proof system when applied to concurrent programs is that almost no predicate can be guaranteed to hold upon a statement's completion if another statement could be executing concurrently. For example, if the statement $x := 0$ was executed concurrently with $x := x + 1$ then we could not conclude $\{x \geq 0\}x := x + 1\{x \geq 1\}$. To allow the partial correctness proofs for each task in a concurrent program to be combined into a partial correctness proof of the whole program, Owicki and Gries defined a condition for proofs of concurrently executing tasks to be non-interfering, essentially that a statement of one task cannot change the truth value of a precondition or postcondition of a statement of another task. Lamport [47] proposed an approach for proving the correctness of concurrent programs that introduces a notation for describing when actions precede or can influence other actions. Dillon [30] has a method for verifying general safety properties of concurrent Ada programs that combines partial correctness proofs for each of the tasks and a global proof of cooperation to account for intertask communication.

Verification of real-time systems has recently received much attention. Shaw [69] has extended Hoare logic to reason about execution times by adding a variable to represent the current time. The execution of an atomic statement increments this variable by the duration of the statement, and the proof system has been augmented to calculate the execution time of a composite statement from the execution times of the statements it comprises. Jahanian and Mok [44] use a decision procedure based

on Pressburger arithmetic to verify safety properties of a real-time system. Ghezzi, Mandriolli, and Morzenti [35] use a real-time logic called Trio to specify and verify properties of real-time systems. Lynch and Attiya [52] specify both real-time systems and their timing requirements as timed I/O-automata and prove the system satisfies its requirements by constructing a mapping from the requirement automaton to the implementation automaton. Davies *et al* [27] add time to CSP and present restrictions of these timed processes that make them well behaved under composition.

2.2 Concurrency Analysis

In this section, I focus on work most closely related to my own, namely analysis methods for concurrent software. All of these methods use some type of reachability search of the kind described by Taylor [72], but they vary in the way they reduce the complexity of the search. All approaches are likely to be intractable in the worst case since the most basic concurrency questions for finite state processes with synchronous communication (e.g., does the system deadlock, does a particular rendezvous occur) are NP-complete [71]. If processes communicate via unbounded FIFO buffers, then most of the interesting questions are undecidable [67].

Though its size is generally exponential in the number of processes, the full reachability graph of a concurrent system contains all the information needed to answer most concurrency questions about the system. If this graph can be constructed, many interesting properties of the system it represents can be decided easily. Clarke, Emerson, and Sistla [16] give an algorithm for deciding whether a concurrent system satisfies an arbitrary assertion of branching-time temporal logic. The running time of this algorithm is linear in the size of the reachability graph of the system and linear in the size of the logic formula. Lichtenstein and Pnueli [50] give an analogous model-checking procedure for linear time temporal logic. Their algorithm has running time linear in the size of the reachability graph of the system but exponential in the size of the logic formula. Aggarwal, Courcoubetis, and Wolper [2] present a method

for specifying and verifying liveness properties of a concurrent system in the context of a reachability graph analysis. For example, one might want to verify that a message is received infinitely often given that the specification requires that infinitely many messages are sent and a message can be lost only a finite number of times. This work extends the work of Clarke, Emerson, and Sistla by allowing more sophisticated fairness requirements for legal computation paths. Karam and Buhr [45] show how a modified reachability graph that they call a *PSA-tree* can be used to detect deadlock, critical races, and starvation in an Ada program. Ostroff [60] analyzes concurrent real-time systems by encoding enough timing information in the states to enforce the time bounds on the transitions of his *timed transition systems.*

I have divided the analysis methods into four categories based on how they attempt to reduce the complexity of searching the reachability graph of the concurrent system. The methods in each of these four categories are described in the following sections. The first collection of methods attempts to reduce the size of the reachability graph by eliminating redundant interleavings. The second set of methods uses necessary conditions to find all potential executions that could have a property and restricts further search to these executions. The third set of methods uses sufficient conditions to find some of the executions that have a property. The fourth set of methods attempts to reduce the complexity of the analysis by building the reachability graph compositionally.

2.2.1 State Space Reduction Approaches

The full reachability graph of a concurrent system represents all legal interleavings of the actions of the component processes. Many of these interleavings might have the same concurrency properties. For example, suppose that one interleaving contains two consecutive reads of a global variable from different processes. Exchanging the order of these two actions cannot change the behavior of the system, thus this new interleaving is equivalent to the original. On the other hand, if a read and a write of a

global variable are transposed, the behavior of the concurrent system could change, so exchanging these two actions would not produce an equivalent interleaving. A notion of equivalence is used to partition the set of interleavings into equivalence classes that preserve the concurrency properties of interest (i.e., either all the interleavings in a class have a certain property, or none do). The techniques in this section attempt to reduce the size of the reachability graph by considering only one interleaving from each equivalence class. The state space of a reachability graph that contains only a subset of the legal interleavings is usually smaller than the original and is called a *reduced state space.*

The simplest state space reduction technique is known as *virtual coarsening* and was first used in [4]. The idea is based on the classification of a process's actions as internal or external such that only the relative order of external actions is significant in determining the behavior of the concurrent system. For example, in an Ada tasking program, assignments to local variables would be internal actions, whereas assignments to global variables and rendezvous would be external actions. Two interleavings are considered equivalent if the relative order of their external actions is the same. Interleavings that differ only in the order of two internal actions must have the same concurrency properties. One way to generate only one interleaving in an equivalence class is to use coarser actions that consist of sequences of the original actions that contain exactly one external action. For example, suppose we have two tasks, each of which has ten actions executed in sequence, only one of which is external. Assuming the external actions are nonblocking, the full reachability graph would contain $\frac{20!}{10!10!} = 184,756$ interleavings, but by coarsening the actions of each task into one virtual action, we would need to examine only $\frac{2!}{1!1!} = 2$ interleavings (one for each order in which the external actions could occur). Long and Clarke [51] have developed such a coarsened representation of Ada programs called *task interaction graphs.* In addition, many other state space generators use this as an optimization (e.g., [8], [54]). In most systems I have examined, this technique significantly reduces

the number of interleavings that must be examined, but still leaves exponentially many. It has the effect of reducing the base of the exponential function (e.g., instead of examining $\sim 10^n$ interleavings of n tasks, we must examine $\sim 4^n$). The reduced reachability graph generated with this technique preserves all concurrency properties of interest.

A more sophisticated notion of equivalence is found in a set of similar techniques that use partial orders. These techniques separate the state explosion due to choices made by the tasks from the state explosion due to different equivalent interleavings of one set of choices, and can largely eliminate the latter. All of these techniques can establish equivalence between interleavings in which external actions occur in different orders as long as these actions produce the same partial order of actions. The technique of Valmari [75] uses the idea of a *stubborn set* of transitions, a subset of the transitions currently enabled whose exploration will generate at least one interleaving for each partial order of the actions. In the case of the dining philosophers problem, this method reduces the size of the state space from exponential to quadratic, allowing huge systems to be analyzed. The original technique focused on deadlock detection, but Valmari has recently extended the work to allow the verification of linear temporal logic assertions from the reduced graph [77]. Godefroid and Wolper [36] have a similar technique using *sleep sets* that they claim are easier to compute than stubborn sets and still yield good reductions. Probst [63] seems to use partial orders for this purpose as well.

Shatz, Tu, and Murata [74] use Petri nets to analyze Ada tasking programs. They translate an Ada program into a Petri net, construct the reachability graph of the Petri net, and then use this graph to answer questions about the original program. Before generating the reachability graph, they use reduction rules to transform the Petri net into an equivalent one with fewer places and transitions. This simpler net has a smaller reachability graph in which many interleavings of the original net's transitions have been eliminated. Like virtual coarsening, this technique seems to

leave an exponential number of interleavings to examine, but there is insufficient experimental data to evaluate the method at this time.

McDowell [54] considers the case where many of the tasks follow identical instructions and gives a state space reduction that takes advantage of this symmetry. The basic idea of the technique is that if a concurrent program has n identical tasks each of which could be in one of k states, then we need not keep track of which tasks are in which states but only how many tasks are in each of the k states. This technique can drastically reduce the size of the reachability graph for systems with a large number of identical tasks.

A more general exploitation of symmetry is found in the work of Huber *et al* [43] in which isomorphic subgraphs of a reachability graph are combined. In the dining philosophers problem, for example, there is a rotational symmetry between the philosophers: the subgraph reachable after philosopher 1 picks up his/her fork is isomorphic to the subgraph reachable after philosopher 2 picks up his/her fork. Thus, we could construct a compressed reachability graph in which the first transition represents any one of the philosophers picking up his/her fork and successive transitions represent, not an action of a specific philosopher, but rather an action of a philosopher n steps to the right of the first philosopher to pick up his/her fork. Symmetries can be verified automatically, but usually must be provided by the analyst. Since the time to check if a state is symmetric to a previously generated state is significant, this method usually does not result in large time savings over standard reachability analysis, but the size of the compressed graph can be much smaller than the full reachability graph and so space and further time spent processing the graph can be saved [65].

Burch *et al* [15] have a technique called *symbolic model checking* for verifying that a concurrent system meets a set of temporal logic specifications. Their technique uses binary decision diagrams (BDDs) to represent the relations and formulas of the logic compactly. This representation captures certain kinds of regularity in the state space

very well. They give an example of a pipelined adder in which the size of the BDD grows linearly with the width of the registers, while the number of states grows exponentially.

Clarke *et al* [17] have a technique they call *abstraction* that uses homomorphisms to abstract away detail in the state space that is not important for the verification of a specific property. Although these abstractions must be provided by the analyst and vary with the property being verified, they allow certain properties of extremely large systems to be verified with model checking.

2.2.2 Necessary Condition Approaches

The methods in this section use conditions that are necessary but not sufficient for a concurrent system to possess some property (usually deadlock). As a result, if the concurrent system under analysis does have the property, then these methods will never report that it does not, but if the concurrent system does not have the property, then these methods may give a spurious report that it does. Thus all reports that the system has the property must be examined by other methods (possibly by hand) to determine if they are spurious.

Young and Taylor analyze Ada tasking programs by ignoring the program variables and their effect on the flow of control. Whenever a task reaches a conditional statement, it is assumed to nondeterministically choose which path to follow. Using this assumption, each task becomes a finite state processes whose state represents the statement to be executed next. Spurious deadlock can be reported if the ignored variable values would have precluded one or more of the conditional branches leading to the deadlock state. Once a suspected deadlock is discovered, symbolic execution is used to determine if the deadlock state is reachable [80]. The reachability graph generated can also be used to verify temporal logic assertions [81], but since this graph contains a superset of the execution paths in the full reachability graph (that reflects the effect of program variables on flow control), only statements about paths

not existing can be verified. This technique has been implemented as part of the CATS toolset described in [81].

Murata, Shenker, and Shatz [58] use Petri nets to find deadlocks in Ada tasking programs. They divide deadlocks into two categories: *inconsistency deadlocks* and *circular deadlocks*. Their method for finding circular deadlocks is a necessary condition approach. After translating the Ada tasking program into a Petri net, they search for certain types of cycles in the net that are necessary for circular deadlock. Once such a cycle is found, they use a reachability search to determine if the cycle, and the deadlock it represents, can be reached. The *T-invariants* of the Petri net, essentially transition counts for possible firing sequences of the net, are used to reduce the complexity of this search.

Masticola and Ryder [53] have a similar categorization of deadlocks into those that are circular, which they call *deadlocks*, and those that are not, which they call *stalls*. They use a similar technique to find circular deadlocks by searching the program's *synch graph* for cycles that meet necessary conditions to represent deadlocks. They have polynomial time algorithms for detecting these cycles and claim that their conditions eliminate most non-deadlock cycles in the experiments they have run, although they admit that most of the programs tested had a very simple tasking structure. They do not address detection of stalls.

Constrained expressions [8], which will be described in Chapter 3, is also a necessary condition approach.

2.2.3 Sufficient Condition Approaches

Whereas necessary condition approaches are overly conservative, not allowing any executions possessing the property to escape detection, sufficient condition approaches are reckless and may not detect all executions having the property. These approaches use conditions that, if satisfied, guarantee the existence of an execution possessing the property, but, if not satisfied, do not guarantee the absence of any executions

with the property. As in the case of the necessary condition approaches, the property usually checked for is deadlock.

Murata, Shenker, and Shatz [58] use Petri net invariants to find certain types of deadlocks in Ada tasking programs. Once the Ada tasking program has been translated into a Petri net, a spanning set of T-invariants is found for the Petri net. Every deadlock-free execution of the Ada tasking program has a corresponding T-invariant, so if a statement does not occur in any T-invariant of the net, then that statement cannot be executed in any deadlock-free execution of the Ada program. Thus execution of that statement will cause deadlock. This is only a sufficient condition for deadlock because not every T-invariant corresponds to a deadlock-free execution of the Ada program. This technique only detects what they call inconsistency deadlocks. The remaining deadlocks, called circular deadlocks, are detected using another method described in the previous section.

Holzmann [42] uses a technique that is very fast in practice. The slowest operation in a reachability search, he maintains, is the check to see if a state has been generated before. He uses hashing to perform this operation approximately, but very quickly. Collisions in the hashing can result in a false report that a state has been visited before. Since the state space is usually strongly connected, however, it is unlikely that pruning some paths will make a deadlock state unreachable. He claims very good coverage on spaces having 10^6 to 10^7 states, which can be searched on a minicomputer in a matter of minutes. This technique amounts to searching part of the reachability graph for an anomaly as quickly as possible.

2.2.4 Compositional Approaches

The approaches discussed in this section attempt to reduce the complexity of the analysis by composing the components of a concurrent system in stages and hiding internal details of the composed entity after each stage. These methods will perform well whenever there are subsets of system components whose interaction with the rest

of the system is simple (i.e., the subsystem could be replaced by a small finite state process that would be indistinguishable to the rest of the system).

Most compositional approaches are based on the notion of a *process algebra*, a formal system in which the behavior of processes can be specified and in which processes that are equivalent in some behavioral sense can be proven so with a set of axioms. There have been many process algebras proposed including Milner's CCS [55, 56], Hoare's CSP [41], Bergstra and Klop's ACP [14], and Hennessy's EPL [39].

As an example, a process that can either perform actions a and then b, or a and then c might be written $ab+ac$. The choice operator $+$ is always commutative, so we would have an axiom $x + y = y + x$ that would allow us to prove the above process equivalent to $ac + ab$. Depending on the notion of equivalence, we may or may not have a distributive law $xy+xz = x(y+z)$. The process $a(b+c)$ will perform the same sequence of actions as $ab+ac$, but since the choice is made before the a in $ab+ac$, this process can refuse to perform one of b or c after doing an a, but the process $a(b+c)$ must perform whichever of b or c is requested after an a.

Most process algebras have an internal, invisible, or silent step, usually represented by the symbol τ. After two processes are composed, any communication involving these two processes is usually hidden by replacing the communication symbols by τ. Axioms in the process algebra allow the resulting process to be simplified. For example, ACP [14] does not distinguish between a process performing a visible action and the process performing the same visible action followed by an invisible action: $ax = a\tau x$.

Another important aspect of process algebras is their notion of process equivalence. Several have been proposed, including trace equivalence [55], failure equivalence [41], and observational equivalence [55]. The stronger equivalences, like observational equivalence, are based on the notion of a bisimulation: two processes are equivalent if they can each simulate the other in some formal sense. Weaker

equivalences focus on the possible sequences of communications in which a process could engage or refuse to engage.

Simulation also leads to the idea of a preorder among processes. If process A can simulate process B but not vice versa, then process B is in some sense more deterministic than process A. If process A is thought of as a specification of a component's behavior, and process B is the implementation of the component (composed from other components), then we want to require that process A can simulate process B (all behaviors of process B are allowed by the specification), but not necessarily that process B can simulate process A.

Valmari [76] describes a framework for compositional analysis that integrates state reduction techniques with the composition. Zuidweg [82] describes an analysis tool for ACP that supports composition, hiding, and testing for bisimilarity. Cleaveland's Concurrency Workbench tool [19], based on Milner's CCS, supports these basic operations plus preorder checking. Clarke *et al* [18] have extended their work on model checking to take advantage of compositional analysis. Yeh and Young [79] have constructed a tool for compositional analysis of Ada-like specifications.

Chapter 3

Constrained Expressions

The contributions described in the sequel were done as part of the constrained expression project. This chapter gives an overview of the constrained expression project, presenting work done by the author together with George Avrunin, Ugo Buy, Laura Dillon, and Jack Wileden. I will use "we" when describing work done jointly. Constrained expressions is primarily an analysis technique for concurrent systems. It is based on an expressive formalism and has been automated by a prototype toolset. Section 3.1 describes the formalism, Section 3.2 presents the analysis technique, and Section 3.3 gives an overview of the toolset that automates the technique.

3.1 Formalism

The constrained expression formalism models a concurrent system as a collection of coupled finite state automata (FSAs) with additional constraints expressed as a set of recursive languages on the alphabets of the FSAs. The acceptance of a symbol by an automaton represents the occurrence of an event in the concurrent system. An event may represent a normal action of a component, such as initiating a communication with another component, or an error, such as waiting forever for a communication that never takes place. An execution of the concurrent program is thus modeled by a string of event symbols.

The following definitions are required to define a constrained expression representation of a concurrent system. For any sets of symbols Σ and S with $S \subseteq \Sigma$, let

$\rho_S : \Sigma^* \rightarrow S^*$ be the homomorphism, called *projection on* S, defined by extending the map $\Sigma \rightarrow S$ given by:

$$\rho_S(\alpha) = \begin{cases} \alpha & \text{if } \alpha \in S \\ \lambda & \text{otherwise} \end{cases}$$

Let $L(M)$ be the language recognized by machine M. The *shuffle* of the languages L_1 and L_2, written $L_1 \otimes L_2$, is the language consisting of the strings $x_1y_1x_2y_2 \ldots x_ny_n$ formed by concatenating substrings such that $x_1x_2 \ldots x_n \in L_1$ and $y_1y_2 \ldots y_n \in L_2$ for some n (the substrings x_1 and y_n may be empty). Finally, let *dagger* (†) be the closure of the shuffle operation:

$$L^\dagger = \{w | \text{for some } n\text{: } w = w_1 \otimes w_2 \otimes \ldots \otimes w_n, w_i \in L \text{ for all } i\}$$

A constrained expression representation of a concurrent system is a triple (M, C, T) where M is a set of FSAs $M_1, \ldots, M_n$ with alphabets $\Sigma_1, \ldots, \Sigma_n$, $\Sigma = \bigcup_i \Sigma_i$, C is a set of recursive *constraint* languages $C_1, \ldots, C_m$ with alphabets $A_1, \ldots, A_m$, where $A_i \subseteq \Sigma$ for all i, and $T \subseteq \Sigma$ is a terminal alphabet. A string $t \in T^*$ represents a legal behavior or trace of the concurrent system if there exists a string $s \in \Sigma^*$ with $\rho_T(s) = t$ where $\rho_{\Sigma_i}(s) \in L(M_i)$ for all i and $\rho_{A_j}(s) \in C_j$ for all j. An *augmented trace* is an alternating sequence of states and events $s_1, e_1, s_2, e_2, \ldots s_k$ where:

- Each s_i is *global state* represented by an n-tuple $< r_{i,1}, \ldots, r_{i,n} >$ where $r_{i,j}$ is a state of M_j.
- For all $j = 1, \ldots, n$, $r_{1,j}$ is the starting state of M_j.
- For all $j = 1, \ldots, n$, $r_{k,j}$ is an accepting state of M_j.
- For all $i = 1, \ldots, k-1$, and all $j = 1, \ldots, n$, if $e_i \in \Sigma_j$ then M_j has a transition from $r_{i,j}$ to $r_{i+1,j}$ on e_i and otherwise $r_{i,j} = r_{i+1,j}$.
- For all $j = 1, \ldots, m$, $\rho_{A_j}(e_1e_2 \ldots e_{k-1}) \in C_j$.

An example of the use of this model to describe a system of two tasks communicating by asynchronous message passing is given in the next section.

The constrained expression formalism originated in the doctoral thesis of Jack Wileden [78], who defined the initial formulation of constrained expressions, wrote translation rules for mapping system descriptions written in a subset of the modeling language DYMOL into this representation, and showed how this representation could be used for analysis. Later, Laura Dillon [29] extended the formalism to represent prefixes of traces and also showed how different formalisms, such as CSP and Petri nets, could be represented in constrained expressions. These earlier formulations contained various restrictions which have been removed from the current formulation for simplicity. For example, earlier formulations required the alphabets of the FSAs to be disjoint and required that the constraint languages be only those generated by the standard regular operators (concatenation, union, and Kleene star) plus shuffle and dagger. It follows from [3] that all recursively enumerable languages are given by such a restricted formulation. Therefore, removing these restrictions to yield the current formulation does not change the expressive power of the formalism.

3.2 Analysis Technique

The main constrained expression analysis technique uses integer linear programming (ILP) to prove properties of concurrent systems. Given a concurrent program represented in the above model, we generate a system of linear inequalities reflecting much, but not all, of the semantics of the representation to determine if any executions of the concurrent program exist that satisfy certain properties. Informally, the method finds a possible execution of the concurrent program by finding traces of each task and then enforcing a weaker consistency criterion between these traces than is specified in the constraint languages.

When generating equations for the FSAs $M_1, \ldots, M_n$, it is useful to picture each FSA as a directed graph in the standard way. An execution of the concurrent program

will correspond to a path through each FSA from the start state to an accepting state. We assign a *transition variable*, x_k, to each transition arc k in the FSAs of the concurrent program. The variable associated with an arc will represent the number of times that arc is crossed in the paths. We also assign an *accept variable*, f_j, to each accepting state j of the FSAs that will be one if and only if the path through that FSA ends at this accepting state and will be zero otherwise. We then generate a *flow equation* for each state j in an FSA stating that the flow into the state (i.e., the total number of times the path enters the state) must equal the flow out of the state (i.e., the total number of times the path leaves the state). The start state of each FSA has an implicit flow in of one, and each accepting state has an extra flow out represented by its accept variable. The flow equations imply that, in any nonnegative integer solution, exactly one accept variable in each FSA will have the value one.

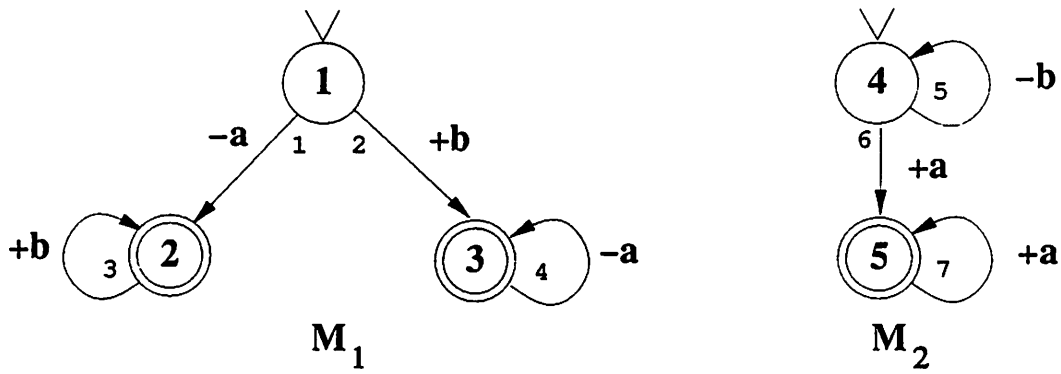
Suppose that a symbol α belongs to two alphabets Σ_i and Σ_j. Then an occurrence of α in a trace represents the occurrence of an event in the tasks corresponding to M_i and M_j. In this case, we add an equation stating that the numbers of occurrences of α in the traces of those tasks are the same. In other words, we equate the sum of the transition variables for arcs of M_i labeled by α with the corresponding sum for M_j.

In addition to the equations generated in this fashion from the FSAs $M_1, \ldots, M_n$, we generate equations and inequalities reflecting the restrictions imposed by the C_j. If a constraint language C_j is regular, we can generate equations from an FSA accepting it, exactly as described above. Then, for each symbol $\alpha \in A_j$ and each i such that $\alpha \in \Sigma_i$, we add an equation stating that the sum of the variables for arcs labeled by α is the same in the FSA accepting C_j as in M_i, just as for symbols belonging to two FSA alphabets. In practice, the constraint languages used by the tools are simple enough that the additional inequalities can be expressed directly in terms of the variables from the M_i, avoiding the creation of many new variables and equations and reducing the size of the ILP problems that must be solved. (An example of this is given below.) There has been no attempt to formalize a procedure for efficiently generating

inequalities from arbitrary recursive languages, but we have devised methods to generate inequalities from constraint languages modeling synchronous and asynchronous communication. Since most other communication mechanisms (e.g., global variables, monitors) can be modeled with additional tasks and synchronous communication, our current technique should be sufficient for analyzing a wide variety of concurrent systems.

Figure 3.1 shows the equations for a simple concurrent program with two tasks that use channels with unbounded message buffers to communicate. Here, $+a$ represents the sending of a message to channel a and $-a$ represents the reception of a message from channel a. Consistent communication over a channel a is enforced by the constraint language $(+a\ -a)^\dagger \otimes (+a)^*$ (this expression generates strings having the property that, in any prefix, the number of $-a$'s never exceeds the number of $+a$'s). From this we extract the relation that the number of $+a$ event symbols must be greater than or equal to the number of $-a$ event symbols in any string representing an execution.

Every trace will correspond to some solution to the inequality system. However, not all solutions to the inequality system will correspond to traces. The conditions represented by the inequality system are thus necessary, but not sufficient. There are two reasons for this. First, information about the order of event symbols in the constraint languages is ignored. In the example of Figure 3.1, there is no execution in which task 1 executes $-a, +b$ ($x_1 = x_3 = f_2 = 1,\ x_2 = x_4 = f_3 = 0$) and task 2 executes $-b, +a$ ($x_5 = x_6 = f_5 = 1,\ x_7 = 0$) since any interleaving of these two strings violates the constraint for channel a or b, but since the number of events is consistent with relations derived from the constraints, these flows are a solution to the inequality system. The second reason that the conditions are not sufficient is that the flow equations do not completely capture the semantics of the FSAs. In a given execution, the events occurring in each task must lie along a single path through that task's automaton, but the presence of cycles in the FSA allows extra



Flow:			**(state)**
1	$=$	$x_1 + x_2$	(1)
$x_1 + x_3$	$=$	$x_3 + f_2$	(2)
$x_2 + x_4$	$=$	$x_4 + f_3$	(3)
$1 + x_5$	$=$	$x_5 + x_6$	(4)
$x_6 + x_7$	$=$	$x_7 + f_5$	(5)
Communication:			**(channel)**
$x_1 + x_4$	$\leq$	$x_6 + x_7$	(a)
x_5	$\leq$	$x_2 + x_3$	(b)

Figure 3.1. Example and its Inequality System

cyclic flows. An example of this is the solution to the inequalities of Figure 3.1 in which $x_1 = x_4 = x_6 = x_7 = f_2 = f_5 = 1$ and all other variables are zero.

The analyst searches for traces with certain properties by adding additional inequalities to the system. For example, an analyst might ask whether there is an execution in which more messages are sent to channel b than are received from that channel by adding the inequality $x_2 + x_3 > x_5$. Similarly, if we added transitions labeled with symbols representing permanent blocking of the task, the analyst could seek executions in which a task waits forever to receive a message by adding an inequality stating that the sum of the transition variables labeled with such a symbol is greater than or equal to one. This type of inequality, used to search for deadlock, is the most commonly added inequality in the analyses described in [8].

I now describe how we model synchronous communication and how the above analysis technique is applied to concurrent systems that use this type of communi-

cation mechanism. From this point on, I will assume a synchronous communication mechanism that allows pairs of tasks to synchronize on actions. This synchronization is assumed to be the only means of inter-task communication. In the sequel, I will present extensions to this technique by describing what inequalities would be generated for this type of concurrent system. All of these extensions can also be used with an asynchronous communication mechanism, but I will only sketch the slight modifications needed to apply the new techniques to systems with asynchronous communication.

We model synchronous communication as follows. Conceptually, a communication occurs through a channel that connects a pair of tasks. This communication is represented in the FSAs of these tasks by either a single symbol appearing in the alphabets of both FSAs (as is used in most of the examples in this dissertation), or by a pair of matching symbols in each task (as is used in the toolset to model our Ada-based design language). In the latter case, a communication over channel c is represented in one task by the pair of events `call`(c) and `resume`(c) in sequence, and in the other task by the pair of events `beg_rend`(c) and `end_rend`(c) in sequence. These two sequences are forced to overlap by the constraint language

$$(\texttt{call}(c)\texttt{beg\_rend}(c)\texttt{end\_rend}(c)\texttt{resume}(c))^*$$

Using a pair of symbols in each FSA allows nested synchronizations (e.g., rendezvous within Ada accept bodies) to be modeled, though I will not use this feature of the design language in this dissertation.

The asymmetry of the communication mechanism in the Ada-based design language is mirrored in the automaton representation of the tasks. One of the tasks connected by a channel is designated as the *caller* and the other as the *acceptor*. In a given state, a task either acts as a caller or as an acceptor. As in Ada, a task may be ready to act as a caller for one channel, but it may be ready to act as an acceptor for any number of channels (representing multiple callers of a single entry or the Ada

`select` statement). If communication is represented by pairs of matching symbols in the automata, then the automaton containing the `call(`c`)` symbol is the caller. Either way the synchronous communication is modeled, for each task and channel, there is a *hang symbol* that represents the "event" that the task waits forever for a communication on that channel. For each channel c, there is a hang symbol for the caller, denoted `hang_c(`c`)`, and one for the acceptor, denoted `hang_a(`c`)`. These symbols will usually be abbreviated `h_c(`c`)` and `h_a(`c`)` when they appear in figures showing automata.

For systems with synchronous communication, the flow equations are generated in the same way as in the example of Figure 3.1, but the communication inequalities are different. For each channel c connecting tasks A and B:

1. A *synchronization equation* requires that the number of times task A communicates over channel c equals the number of times task B communicates over channel c.

2. A *hang inequality* requires that at most one of the tasks A and B becomes permanently blocked waiting for a communication on channel c.

The algorithm for generating inequalities from FSAs is shown in Figure 3.2. For a state j, *in(j)* is the set of transitions into the state and *out(j)* is the set of transitions out of the state. For a channel c connecting task A (the caller) to task B (the acceptor), *call(c)* is the set of transitions of task A labeled with synchronous communication on c, while *accept(c)* is the set of transitions of task B labeled with synchronous communication on c. Also, *hang_c(c)* is the set of transitions of task A labeled with a hang symbol for communication on c, while *hang_a(c)* is the set of transitions of task B labeled with a hang symbol for communication on c. The notation $[expr]_{cond}$ indicates that the expression *expr* should be added to the inequality only if the condition *cond* is true, and otherwise zero should be added. The existence of every variable appearing in *expr* is always implicitly conjoined to this condition, which defaults to true if not

Input: A set M of FSAs
Output: A set of inequalities

For each transition k in an FSA of M:
  Create transition variable x_k
For each accepting state j of an FSA of M:
  Create accept variable f_j
For each state j of an FSA of M:
  Generate flow equation: $[1]_{start(j)} + \sum_{k \in in(j)} x_k = \sum_{k \in out(j)} x_k + [f_j]$
For each channel c:
  Generate synchronization equation: $\sum_{k \in call(c)} x_k = \sum_{k \in accept(c)} x_k$
  Generate hang inequality: $\sum_{k \in hang\_c(c)} x_k + \sum_{k \in hang\_a(c)} x_k \leq 1$

Figure 3.2. Basic Algorithm

specified. This construct is useful to include accept variables in flow equations only when they exist (i.e., for accepting states) by using $[f_j]$ in the flow equation for state j. Also, the implicit flow of one into start states may be indicated by using $[1]_{start(j)}$ in the flow equation for state j where *start(j)* is a predicate that is true if j is a start state of its automaton. A table of the abbreviations used in pseudo-code descriptions of the inequality generation algorithms is given in Appendix A. In general, j, j' will range over states, k, k' will range over transitions, and c will range over channels.

We would generate inequalities for systems with an asynchronous communication mechanism as follows. For a given channel c, let r_c be the number of message receptions on that channel and s_c be the number of message sends on that channel. If the channels are unbounded buffers, then for each channel c we generate the single communication inequality: $r_c \leq s_c$. If the channels are bounded buffers of length N, then for each channel we generate the pair of communication inequalities: $r_c \leq s_c$ and $s_c \leq r_c + N$.

Finally, we note that the flow equations can also be generated from regular expressions (REs) or from a hybrid representation of regular languages called *regular expression deterministic finite automata* (REDFAs), which are essentially finite automata whose arcs are labeled with regular expressions. REDFAs yield particularly compact systems of inequalities [20]. The generation of inequalities from REs and REDFAs is described in [8].

3.3 Toolset

The analysis technique described in the last section has been implemented as part of the *constrained expression toolset.* A version of the toolset is described in detail in [8], along with an extensive series of experiments that were conducted with it. That version was modified to conduct most of the experiments with the new techniques described in this dissertation. This section gives an overview of the toolset detailed in [8]. Section 7.1 describes how the toolset was modified for the experiments described in the sequel.

There are five major components of the constrained expression toolset (see Figure 3.3). In normal use, an analyst first uses the *deriver* to produce a constrained expression representation from a concurrent system design written in the CEDL design language. This constrained expression is then used as input to the *constraint eliminator*, which modifies the representation for reasons explained below. The *inequality generator* takes the constrained expression produced by the eliminator as its input, together with a query formulated by the analyst, and produces a system of linear inequalities representing necessary conditions for the existence of a trace of the concurrent system satisfying the query. The IMINOS *integer programming package* is used to determine whether this system has any integral solutions and, if it does, to find one with appropriate properties. If a solution is found, the *behavior generator* uses heuristic search techniques to determine whether this solution corresponds to a trace of the system, and to produce such a trace if it does.

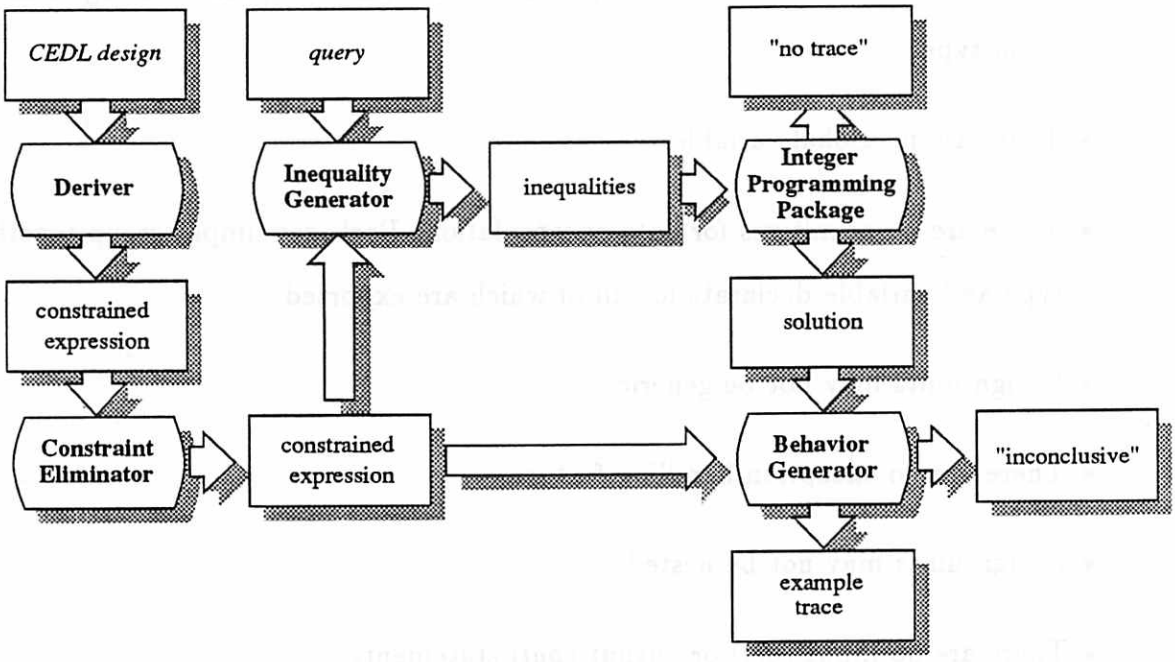


Figure 3.3. Diagram of Constrained Expression Toolset

In the remainder of this section, I discuss each of the components of the toolset in more detail. Note that the constraint eliminator tool and the behavior generator tool process DFAs, while the inequality generator tool, like the analysis technique it automates, can handle FSAs (NFAs and DFAs).

3.3.1 The Deriver

The deriver [5] provides a front-end for the constrained expression toolset. It translates system designs into constrained expressions, which are then manipulated and analyzed by various other tools.

The current deriver requires that designs be written in CEDL, an Ada-like design language. CEDL focuses on the expression of communication and synchronization in a concurrent system, and language features not related to concurrency are kept to a minimum. The most important limitations of CEDL designs can be summarized as follows:

- *Boolean* is the only predefined type; all other types are specified using enumeration types.
- There are no global variables.
- There are no primitives for data encapsulation. Packages simply group together type and variable declarations, all of which are exported.
- Design units may not be generic.
- There are no exception handling features.
- Design units may not be nested.
- There are no input (*get*) or output (*put*) statements.

Most of the Ada control-flow constructs have correspondents in CEDL. CEDL also provides an ellipsis notation (written "...") for expressing incompleteness in designs. The use of this construct will be illustrated in an example below. The incompleteness construct can be used to elide statements, expressions, declarations and types that will be elaborated in later system descriptions.

The deriver produces a *task expression* for each of the tasks in a CEDL design from the code for the task bodies. The deriver produces the constraints for the constrained expression representation of a CEDL design by instantiating a fixed set of constraint templates. Each task expression or constraint is a regular expression over symbols representing events such as the calling of an entry, the assignment of a value to a variable, etc.

To illustrate the translation performed by the deriver, consider a CEDL version of the dining philosophers problem with three philosophers. Figure 3.4 shows the CEDL code for one fork task and one philosopher task from this system. The fork task loops repeatedly, accepting calls to its `UO` and `DO` entries. The philosopher task loops an indeterminate number of times (as indicated by the elided test in the `while`

```
-- Fork 0

task F0 is
     entry U0;  -- pick up fork 0
     entry D0;  -- put down fork 0
end F0;

task body F0 is
begin
    loop
      accept U0;  -- pick up
      accept D0;  -- put down
    end loop;
end F0;
```

```
-- Philosopher 0

task P0;

task body P0 is
begin
    while ... loop
      ... ;      -- Think
      F1.U1;
      F0.U0;
      ... ;      -- Eat
      F1.D1;
      F0.D0;
    end loop;
end P0;
```

Figure 3.4. Fork and Philosopher Tasks from Dining Philosophers in CEDL

statement), calling the U entries of the fork tasks on its "left" and "right" and then calling the D entries of those tasks. There are two more fork tasks and two more philosopher tasks in the system, with similar designs. Figure 3.5 gives the task expressions produced by the deriver for these two tasks, and Figure 3.6 shows some of the constraints produced by the deriver for this system. The expressions are given in the LISP-like prefix notation used as input to several of the tools, which uses "NIL" to denote the empty string, "SEQUENCE" for concatenation, "OR" for disjunction, and "STAR" for Kleene star. Table 3.1 summarizes the interpretation of event symbols. Note that the permanent blocking of a task indicated by the hang symbols does not presuppose any particular cause for this blocking, which could be due to circular deadlock, termination of other tasks, or other reasons. The synchronization constraint in Figure 3.6 enforces proper synchronization of rendezvous for one of the entries of a fork task. Similar synchronization constraints are required for all entries. The hang constraint in Figure 3.6 ensures that the fork task does not wait forever for a rendezvous with one of the philosophers if a philosopher task is also waiting for the same rendezvous. Similar hang constraints would be generated for each entry. Other types of constraints that do not occur in this example enforce the correct dependence

```
(deftask f0
   ("SEQUENCE" "beg_loop(f00)"
      ("STAR"
         ("SEQUENCE"
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.u0)" "end_rend(p1;f0.u0)")
               ("SEQUENCE" "beg_rend(p0;f0.u0)" "end_rend(p0;f0.u0)"))
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.d0)" "end_rend(p1;f0.d0)")
               ("SEQUENCE" "beg_rend(p0;f0.d0)" "end_rend(p0;f0.d0)"))))
      ("OR"
         ("SEQUENCE" "hang_a(f0.u0)" "stop(f0)")
         ("SEQUENCE"
            ("OR"
               ("SEQUENCE" "beg_rend(p1;f0.u0)" "end_rend(p1;f0.u0)")
               ("SEQUENCE" "beg_rend(p0;f0.u0)" "end_rend(p0;f0.u0)"))
            "hang_a(f0.d0)" "stop(f0)"))))


(deftask p0
   ("SEQUENCE" "beg_loop(p00)"
      ("STAR"
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "call(p0;f1.d1)" "resume(p0;f1.d1)"
            "call(p0;f0.d0)" "resume(p0;f0.d0)"))
      ("OR"
         ("SEQUENCE" "end_loop(p00)" "term(p0)")
         ("SEQUENCE" "hang_c(p0;f1.u1)" "stop(p0)")
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)"
            "hang_c(p0;f0.u0)" "stop(p0)")
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "hang_c(p0;f1.d1)" "stop(p0)")
         ("SEQUENCE" "call(p0;f1.u1)" "resume(p0;f1.u1)" "call(p0;f0.u0)"
            "resume(p0;f0.u0)" "call(p0;f1.d1)" "resume(p0;f1.d1)"
            "hang_c(p0;f0.d0)" "stop(p0)"))))
```

Figure 3.5. Two Task Expressions Derived From the Dining Philosophers Problem

```
(defconstraint SYNCHRONIZATION_1
   ("SEQUENCE"
      ("STAR"
         ("SEQUENCE" "call(p0;f1.u1)" "beg_rend(p0;f1.u1)"
             "end_rend(p0;f1.u1)" "resume(p0;f1.u1)"))
      ("OR"
         "NIL"
         ("SEQUENCE" "call(p0;f1.u1)" "beg_rend(p0;f1.u1)"))))

(defconstraint HANG_1
  ("OR"
    "hang_a(f0.u0)"
    ("STAR"
      ("OR" "hang_c(p1;f0.u0)" "hang_c(p0;f0.u0)" ))))
```

Figure 3.6. Some Constraints Generated by the Toolset from the Dining Philosophers System

Table 3.1. Interpretation of Event Symbols

Symbol	*Associated Event*
beg_loop(L)	Begin execution of loop L
beg_rend(S;T.E)	Task T begins rendezvous with task S at entry E
call(S;T.E)	Task S begins rendezvous with task T at entry E
end_loop(L)	End execution of loop L
end_rend(S;T.E)	Task T finishes rendezvous with task S at entry E
hang_a(T.E)	Task T is permanently blocked waiting to accept a call on entry E
hang_c(S;T.E)	Task S is permanently blocked calling entry E of task T
resume(S;T.E)	Task S finishes rendezvous with task T at entry E
stop(T)	Task T stops execution (abnormal termination)
term(T)	Task T terminates (normally)

of control flow on the values of variables and handle the failure of nested rendezvous. An example of the former type is presented in Figure 3.9.

The deriver is, of course, specific to CEDL. In principle, the other tools could be constructed in a CEDL-independent fashion, and used with constrained expressions produced from any design notation. In fact, as discussed below, the inequality generator and behavior generator rely on certain features of CEDL in order to improve efficiency.

3.3.2 The Constraint Eliminator

As discussed in Section 3.2, the inequalities generated do not express the full semantics of constrained expressions, with the result that there may be solutions to the inequalities that do not correspond to traces. In particular, the inequalities do not express certain restrictions on traces that involve only the order in which certain events occur, rather than the numbers of such events in the traces. In practice, the most significant of these restrictions are those imposed by the constraints that ensure the consistent use of variables in CEDL programs. Without taking such restrictions into account, there would be solutions to the inequality system corresponding to "traces" in which, for example, the `else` branch of an `if` statement is taken even though the Boolean condition of the `if` statement evaluates to `true`. The constraint eliminator [21] is used to modify the constrained expression representations in such a way that the inequalities generated from them exclude such solutions.

To see how the constraint eliminator is used, consider the segment of a task `T` shown in Figure 3.7. Figure 3.8 shows the portion of the task expression for task `T` corresponding to this fragment. This segment should always call exactly one of entries `A` or `B` of task `S`; however, the task expression produced by the deriver permits traces in which both calls are made and traces in which neither call is made. In the full constrained expression representation, the dataflow constraint shown in Figure 3.9 filters out these erroneous strings. The constraint allows any number of

```
flag := ...;
if flag then
  S.A;
end if;
if not flag
then
  S.B;
end if;
```

Figure 3.7. Segment of task T

```
("SEQUENCE"
   ("OR" "def(flag;true)"
         "def(flag;false)")
   ("OR" ("SEQUENCE" "use(flag;true)" "call(T;S.A)" "resume(T;S.A)")
         "use(flag;false)")
   ("OR" ("SEQUENCE" "use(flag;false)" "call(T;S.B)" "resume(T;S.B)")
         "use(flag;true)"))
```

Figure 3.8. Part of the Task Expression for Task T

`def(flag;`*val*`)` symbols, each of which represents the assignment of the value *val* to the variable `flag`. It also allows each `def(flag;`*val*`)` symbol to be followed by any number of `use(flag;`*val*`)` symbols with that particular value, each representing a use of the variable, before the next `def(flag;`*val*`)` symbol. Any string satisfying both the task expression and the constraint will include exactly one of the entry calls.

The constraint eliminator modifies the constrained expression so that each of the resulting task expressions incorporates any constraints involving only symbols from that task (i.e., any string satisfying the new task expression satisfies both the old task expression and the constraints). Figure 3.10 shows the result of incorporating the dataflow constraint for the variable `flag` into the task expression for task T. The inequalities generated from the resulting task expression then reflect the restrictions imposed by the constraint, and do not admit solutions corresponding to violations of that constraint.

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

Figure 3.9. Dataflow Constraint for Local Variable `flag`

```
("OR" ("SEQUENCE" "def(flag;true)" "use(flag;true)" "call(T;S.A)"
                  "resume(T;S.A)" "use(flag;true)")
      ("SEQUENCE" "def(flag;false)" "use(flag;false)" "use(flag;false)"
                  "call(T;S.B)" "resume(T;S.B)"))
```

Figure 3.10. Part of Task Expression After Elimination of Dataflow Constraint

The constraint eliminator takes a set of task expressions and constraints as input. Each constraint whose alphabet involves only symbols from a single task alphabet (an *intra-task* constraint) is incorporated into the task expression it constrains and is then removed. The resulting set of task expressions and constraints is output. The task expressions incorporating their intra-task constraints may be output either as REs, DFAs, or REDFAs.

To incorporate a set of intra-task constraints into a task expression, all the REs involved are converted to DFAs, which are then intersected pairwise. The intersection differs from standard DFA intersection in the following way: At each state of a DFA, implicit self-loops are assumed on all symbols not appearing in the alphabet of that DFA. This allows the DFA representing a constraint to accept symbols not in its alphabet without changing state. Assuming the constraint alphabet is a subset of the task alphabet, the result of the intersection is a DFA that accepts exactly those strings accepted by the original task DFA in which the symbols contained in the intra-task constraints appear in the order required by those constraints. In the case of a dataflow constraint for a local variable, this essentially encodes the value of the variable into the DFA state (where before the state encoded only the syntactic location within the task design), usually increasing the number of states in the task DFA, but

guaranteeing consistent use of the variable. In CEDL, the intra-task constraints are exactly the dataflow constraints since there are no global variables and all other constraints involve more than one task.

Using the intersection procedure described above, the constraint eliminator could, theoretically, intersect all the tasks and constraints, producing one large DFA whose language is the set of legal traces of the concurrent system. While this would prevent violation of all the constraints (not just the intra-task ones), the resulting DFA would be similar to a reachability graph of the concurrent system, and equally large—in the worst case exponential in the number of tasks. It is exactly this state explosion the method seeks to avoid by considering the tasks separately and ignoring some of the dependency between them.

3.3.3 The Inequality Generator

The analysis implemented by the constrained expression toolset involves the generation of a system of linear inequalities expressing features of both the constrained expression representation of the concurrent system being analyzed and a query posed by the analyst. I now describe the inequality generator component of the toolset [6].

The input to the inequality generator consists of a list of tasks. The tasks may be represented as REs, or, following constraint elimination, as DFAs or REDFAs. For each task, the inequality generator produces a collection of equations. It then generates additional inequalities reflecting part of the semantics of certain of the constraints. The generation of equations for the tasks depends only on the basic structure of REs and FSAs, but the generation of inequalities from constraints depends on features of CEDL. In principle, since the CEDL constraints are all REs, the generation of inequalities from tasks and constraints could be accomplished in a uniform manner, as described in Section 3.2. While this would be more consistent with the interpretation of the semantics of constrained expressions given in Section 3.1, the separate procedure adopted in the inequality generator improves the efficiency of

the tool and reduces the size of the systems of inequalities it produces, as discussed below.

I first discuss the generation of the equations from the tasks, and then discuss the generation of the inequalities from the constraints. The inequality generator can produce equations from a task in the form of an FSA, an REDFA, or an RE. I have discussed the generation of equations from an FSA in Section 3.2. Since the generation of equations for REs and REDFAs is not used in the new analysis techniques, I omit discussion of it here and refer the interested reader to [6].

Having produced equations for each task, the inequality generator then begins to generate linear inequalities reflecting some of the constraints. The constraints impose restrictions on the order and number of occurrences of event symbols in traces of the system. The integer programming variables we use only involve the total number of occurrences of symbols (or, more precisely, of traversals of arcs in the finite state automata), and do not reflect the order in which those symbols occur. We therefore wish to extract the information about total numbers of occurrences of event symbols from the constraints.

Note first that the total number of occurrences of a particular event symbol is given by the sum of certain variables in the equations generated from the tasks. Specifically, the number of occurrences of a symbol is given by the sum of the transition variables for those arcs labeled by the symbol. To see how the constraints justify additional inequalities, consider first the synchronization constraint shown in Figure 3.6 (page 41). In any string satisfying this constraint, the number of `call(p0;f1.u1)` symbols must equal the number of `beg_rend(p0;f1.u1)` symbols, and the number of `end_rend(p0;f1.u1)` symbols must equal the number of `resume(p0;f1.u1)` symbols. The inequality generator therefore produces equations involving the sums of variables corresponding to the numbers of occurrences of these symbols. The constraint further requires that the various symbols occur in a specified order, but this fact cannot be expressed in terms of the integer programming variables associated with the

tasks. Similarly, from the blocking constraint of Figure 3.6 and the fact that task expressions produced by the deriver have the property that each task contributes at most one **hang** symbol to a trace, We conclude that the sum of the number of **hang_a**(f0.u0) symbols and the number of **hang_c**(pi;f0.u0) symbols cannot exceed one, for $i = 0, 1$. Other inequalities are obtained from the constraints that deal with the failure of nested rendezvous. (The constraints that enforce the dependence of control flow on data involve only the order in which event symbols occur and not the total number of their occurrences, and are ignored in this part of the analysis. The constraint eliminator takes those constraints into account before inequalities are generated.) As noted above, it would be possible to generate inequalities from a constraint by first generating equations from the RE, as we do for task expressions, and then generating equations stating that the number of occurrences of an event symbol coming from the task in which it appears must equal the number coming from each constraint in which it appears. This approach, though pleasingly uniform and language-independent, would lead to the introduction of many additional variables and equations coming from the constraints. We have therefore chosen to sacrifice some of the language-independence and generate inequalities involving the variables from the tasks directly from the CEDL constraint templates.

We thus generate a system of inequalities reflecting a large part, but not all, of the semantics of the constrained expression representation. Queries about the behaviors of the concurrent system are also expressed in terms of the integer programming variables. For example, an analyst could formulate the statement that a philosopher is permanently prevented from eating as an inequality stating that at least one of certain **hang** symbols occurs (i.e., that the sum of certain variables is greater than or equal to one). Adding this to the system of inequalities obtained from the constrained expression, we would obtain a system reflecting both the constrained expression and the query. If this system has no integral solution, then the CEDL system has no trace in which a philosopher task waits indefinitely for a rendezvous with a fork

task. If there is an integral solution, this does not guarantee that a behavior of the CEDL system exists in which the philosopher task waits indefinitely—we have ignored information about order in generating the inequalities, so the solution may be "spurious" in the sense that it does not correspond to an actual behavior. We can, however, use the event counts obtained from the solution as a guide in searching for a real behavior with the property expressed in the query.

3.3.4 IMINOS

The inequality systems produced by the inequality generator are solved using a branch-and-bound algorithm employing the variable dichotomy scheme first introduced by Dakin [25]. The implementation of this algorithm makes use of the MINOS [68] optimization package to solve LP-relaxations of the integer programming problems. The tool that incorporates the branch-and-bound code and MINOS is called IMINOS (Integer MINOS). IMINOS takes an inequality system and associated objective function in the standard MPS file format as input. This input file is produced by the inequality generator.

3.3.5 The Behavior Generator

If IMINOS has produced a solution to the system of inequalities, the next step is to determine whether that solution corresponds to a trace, or behavior, of the concurrent system being analyzed. This is the principal function of the behavior generator [24]. Given the solution and the constrained expression (a set of task REs, DFAs, or REDFAs along with constraints) as input, the behavior generator will attempt to construct a trace using the information in the solution as a guide. This information consists of total event counts for every event symbol and also includes counts for each arc in the DFA representation of the task — provided that the inequalities for that task were generated from either the DFA or REDFA form of the task, rather than from an RE.

The behavior generator performs a highly constrained reachability search of the global state space of the concurrent system. The global state space is, in general, exponential in the number of tasks, but the information in the solution found by IMINOS severely limits the possible actions of each task, frequently allowing no choices whatsoever, and in practice the search has been quite fast. A global state contains the states of the DFAs for all the tasks and constraints (the behavior generator uses the DFA representation for all tasks and constraints, converting REs to DFAs as necessary) as well as the symbol and arc counts being used to guide the search. These counts represent the remaining number of times a symbol or arc may occur; they are started at the values given by the solution and decremented to zero. Once at zero, a count prohibits its symbol or arc from being taken in any successor of that global state, pruning the search tree. The search starts at the global state in which all task and constraint DFAs are in their start state and all counts to be used are set to the value found by the solution. The global state space is searched depth first until a *final global state* is found in which all task and constraint DFAs are in accepting states and all counts are zero, or until a user specified depth bound is reached. Heuristics, some of which are specific to CEDL, control the order in which successor states are generated and can eliminate some states that cannot lead to a final global state.

If a final global state is found, the list of event symbols allowing the global state transitions to the final global state is a trace of the concurrent system. This string of symbols and a list of each task's actions are written to a file and the analyst may then stop or continue the search for other behaviors. If no trace is found within the given depth bound, then the analyst may extend the depth bound and continue the search from the states along the "frontier" of the space (states at the depth bound). If a solution to the system of inequalities is provided, the state space will be finite (there can be no more symbols than given by the solution) and so failure to find a trace within the depth bound given by the number of events in the solution proves no trace

corresponding to the solution exists. The behavior generator also has facilities that allow the analyst to use the tool more interactively by using only a part (possibly none) of the solution and by modifying the solution to require or prohibit certain event symbols from occurring in the trace. Note, however, that the size of the state space increases rapidly as the amount of information given to the behavior generator decreases.

Chapter 4

Verifying Larger Systems

This chapter is the first of four chapters that describe my contributions. As mentioned in Section 3.2, I will present my analysis techniques by describing the inequalities that would be generated for a concurrent system using synchronous communication, and only briefly comment on how these inequalities would be adjusted for asynchronous communication. In this chapter, I describe a pair of often synergistic techniques that allow certain kinds of large concurrent systems to be analyzed very efficiently. The first technique, described in Section 4.1, can represent R copies of a task in a concurrent system by setting certain coefficients in the inequality system to R, as opposed to generating variables and inequalities for each of the R copies of the task. The second technique, described in Section 4.2, can use an integer programming variable to record the value of a local program variable used as a counter, making constraint elimination for such a variable unnecessary and thus avoiding the state explosion inherent in that operation. Since many of the concurrent systems I have encountered have contained identical tasks and/or variables used as counters, I believe the techniques presented here should be very useful.

4.1 Representing Identical Tasks

This section describes a technique for representing many identical tasks compactly in a constrained expression analysis of a concurrent system. McDowell [54] considers the same problem in the context of a reachability search and gives a state reduction technique to improve the efficiency of the analysis. His technique is based on the observation that, given R identical tasks, each of which can be in one of n states, one

need not keep track of the state of each task, but only the number of tasks in each state. My technique exploits this observation in the context of an inequality-based constrained expression analysis.

Tasks are defined to be *identical* if they follow the same sequence of instructions and if other tasks in the concurrent system do not refer to these tasks by name. In CEDL, two tasks are identical if they have identical task bodies and they contain no `accept` statements (since an entry call must refer to the accepting task by name).

An example of a simple concurrent system with many identical tasks is given in Figure 4.1. This abbreviated CEDL specification is for a system with R identical customer tasks, each of which acquires exclusive access to an (unmodeled) resource from the guard task before using it. Customer tasks may terminate while holding access to the resource. The actual specification would be obtained by replicating the text for the customer task R times and substituting $i = 1, \ldots, R$ to obtain unique task names. The FSAs derived for the guard task and one customer task are shown in Figure 4.2. As with all the FSAs shown for demonstration, these are simplified versions of what the automated tools would produce: rendezvous are represented using a single common symbol, various marker symbols (e.g., `stop`, `begin_loop`) have been omitted, and symbol names have been abbreviated (e.g., `h_c` instead of `hang_c`).

In a normal constrained expression analysis of this system, there would be R copies of the customer FSA shown in Figure 4.2 and the technique would find a flow of one from a start state to an accepting state in each of these R FSAs. Instead, the identical tasks technique represents the R customers using the single customer FSA shown in Figure 4.2 and finds a flow of R from the starting state to the accepting states. Conceptually, the transition variable on an arc of this FSA represents the sum of the R transition variables for the same arc in the R copies of the FSA that would normally be used to represent the R customer tasks.

```
task customer_i;

task body customer_i is
begin
  guard.ACQUIRE;
  while ... do
    -- Use Resource
    guard.RELEASE;
    guard.ACQUIRE;
  end while;
end customer_i;
```

```
task guard is
  entry ACQUIRE;
  entry RELEASE;
end guard;

task body guard is
begin
  loop
    accept ACQUIRE;
    accept RELEASE;
  end loop;
end guard;
```

Figure 4.1. CEDL for Example with Identical Tasks

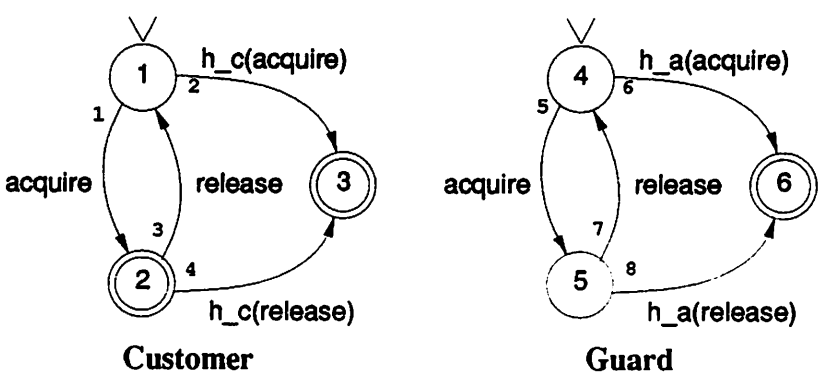


Figure 4.2. FSAs for Example with Identical Tasks

Flow:			**(state)**
$R + x_3$	$=$	$x_1 + x_2$	(1)
x_1	$=$	$x_3 + x_4 + f_2$	(2)
$x_2 + x_4$	$=$	f_3	(3)
$1 + x_7$	$=$	$x_5 + x_6$	(4)
x_5	$=$	$x_7 + x_8$	(5)
$x_6 + x_8$	$=$	f_6	(6)
Synchronization:			**(channel)**
x_1	$=$	x_5	(*acquire*)
x_3	$=$	x_7	(*release*)
Hang:			**(channel)**
$x_2 + Rx_6$	$\leq$	R	(*acquire*)
$x_4 + Rx_8$	$\leq$	R	(*release*)

Figure 4.3. Inequality System for Example with R Identical Tasks

The inequality system for the R customer version is shown in Figure 4.3. This is the same inequality system as would be generated for the one customer version, with two exceptions. First, the start state of the customer task has an implicit flow in of R rather than one. Second, the hang inequalities have to be changed to allow up to R **hang_c** symbols or one **hang_a** symbol, but not both. Given that x_c represents the number of **hang_c** symbols and x_a represents the number of **hang_a** symbols, the inequality $x_c + Rx_a \leq R$ accomplishes this by forcing $x_c = 0$ if $x_a = 1$ and forcing $x_a = 0$ if $x_c \geq 1$. These inequalities can always be changed in this way since only the calling task can be replicated by the technique. The algorithm for generating inequalities in which task i has R_i identical copies is given in Figure 4.4. Here, *fsa(j)* is the FSA containing state (or transition) j and *caller(c)* is the FSA designated as the caller of channel c.

In the context of an inequality-based constrained expression analysis, the representation of a replicated task using one FSA is more efficient than the representation of the task with multiple FSAs. By this, I mean that the size of the inequality system generated for the analysis is smaller, and thus, in my experience, easier to solve. I now show that the inequality system generated for this more efficient representation

Input: A set M of FSAs
A replication constant R_i for each task i
Output: A set of inequalities

For each transition k in an FSA of M:
Create transition variable x_k
For each accepting state j of an FSA of M:
Create accept variable f_j
For each state j of an FSA of M:
Generate flow equation: $[R_{fsa(j)}]_{start(j)} + \sum_{k \in in(j)} x_k = \sum_{k \in out(j)} x_k + [f_j]$
For each channel c:
Generate synchronization equation: $\sum_{k \in call(c)} x_k = \sum_{k \in accept(c)} x_k$
Generate hang inequality: $\sum_{k \in hang\_c(c)} x_k + R_{caller(c)} \left(\sum_{k \in hang\_a(c)} x_k \right) \leq R_{caller(c)}$

Figure 4.4. Algorithm for Identical Tasks

is a set of necessary conditions for the existence of a trace of the concurrent system. Let S_1 be the inequality system generated for a system with R_i copies of task i by the basic algorithm in Figure 3.2 (where a task is replicated by feeding the algorithm multiple copies of its FSA) and let S_2 be the inequality system generated for the same system by the algorithm of Figure 4.4 (where tasks are replicated by setting their replication constants). Each transition k in a task with replication constant R has R transition variables, $x_{k,1}, \ldots, x_{k,R}$, in S_1 (one for each copy of its FSA) and one transition variable, x_k, in S_2.

Theorem 4.1 *S_1 has an integral solution only if S_2 has an integral solution.*

Proof. Given a solution to S_1, the assignment of values to variables of S_2 given by

$$x_k = \sum_{i=1}^{R_{fsa(k)}} x_{k,i} \quad .$$

yields a solution to S_2. To see this, note that each inequality in S_2 can be written as a sum of inequalities of S_1 using the above equation to substitute variables of S_2 for

variables of S_1. The flow equation for a state j in S_2 is the sum of the flow equations for the copies of state j in S_1. The synchronization equation in S_2 for a channel c whose caller has replication constant R is the sum of the synchronization equations in S_1 for the R channels $c_1, \ldots, c_R$ that connect the R copies of the caller of channel c with the acceptor. Similarly, the hang inequality for a channel c in S_2 is the sum of the hang inequalities in S_1 for the R channels $c_1, \ldots, c_R$ that connect the R copies of the caller of channel c with the acceptor. □

Since S_1 is a set of necessary conditions for the existence of a trace of the concurrent system, this theorem proves that S_2 is also a set of necessary conditions. Thus the more efficient representation of a replicated task with a single FSA is sufficient to prove that no trace matching a query exists.

The inequality system in Figure 4.3 represents necessary conditions for the existence of a trace of the R customer version. Like the conditions derived by the normal technique, these conditions are not sufficient for the reasons discussed in Section 3.2. Nevertheless, these conditions can be used in a similar way to either find a trace with a certain property or prove that no such trace exists. For example, I can test for the existence of a trace in which two customers terminate while holding the resource by adding the inequality $f_2 \geq 2$. With this inequality added, the inequality system in Figure 4.3 is inconsistent, proving no such trace exists. On the other hand, I could search for a trace in which a customer becomes permanently blocked waiting to acquire the resource by adding the inequality $x_2 \geq 1$. In this case, a solution exists ($x_2 = R - 1, x_1 = f_2 = x_5 = x_8 = f_6 = 1$) that corresponds to a trace in which one of the customers acquires the resource and terminates while the others become permanently blocked waiting to acquire it.

An experiment with this technique used in combination with the technique of Section 4.2 is presented in Section 7.2.1.

4.2 Counter Variables

This section describes a technique for representing the value of a program variable used as a counter with an ILP variable. This allows dataflow information about such variables to be taken into account in the analysis without performing constraint elimination, as described in Section 3.3.2. Constraint elimination encodes the value of a variable in the state of its task FSA and usually increases the number of states in that FSA significantly.

The technique can only represent variables used as counters. A *counter* is an integer variable in a task having a bounded range $0 \ldots n$ to which only the following operations may be applied:

- The variable is initialized to any legal value before the task containing it begins execution.
- The variable may be incremented or decremented by one. No other assignments to the variable are allowed.
- The variable may be tested for equality or inequality to its range limits, but not to any intermediate value.
- Boolean expressions containing tests on the value of counter variables must be false if any of the counter variable tests are false (i.e., tests on the values of counter variables can only be conjoined).

Despite these restrictions, many variables I have encountered in sample concurrent programs have been counters. For example, in the readers and writers example described in [8], both the integer variable used to count the number of readers using the resource and the boolean variable used to indicate if a writer is using the resource are counter variables.

Dataflow information for counters can be incorporated into the analysis without constraint elimination as follows. The value of a counter variable at the end of the

trace can be calculated by taking its initial value, adding to this the number of times it was incremented, and then subtracting the number of times it was decremented. This ignores the possibility that the order of the increments and decrements sent the counter out of range, which will be discussed later in Section 4.2.1. The value of the counter at the end of the execution may have to satisfy certain conditions, which I call *end conditions*, that can be derived from the task. Specifically, an end condition is a predicate on the value of a counter variable at the end of an execution that is attached to a transition into an accepting state of the FSA. A transition into an accepting state should not be allowed if any of its end conditions are false. An end condition represents a restriction on the behavior of the task's FSA that comes from one of its program variables. If constraint elimination had been performed on this variable, then this restriction would be enforced within the resulting (and probably larger) FSA. Using the counter variable technique, I can enforce these end conditions in the analysis by adding inequalities that prevent a transition into an accepting state from being taken if any of its end conditions are false.

Unlike most of the techniques described in this dissertation, the techniques in this chapter require more information about the concurrent system than simply a set of synchronously communicating FSAs representing the tasks. In the identical tasks technique of Section 4.1, I had to specify a replication constant for each task. For the counter variable technique, I must specify information about the program variables to be stored in ILP variables. Since this information comes from the Ada-like specification rather than the FSAs (which have no program variables), I describe how end conditions are generated for the FSAs from the Ada-like specification that was used to build these FSAs.

I use two types of end conditions in this analysis. The first involves the case where a task becomes permanently blocked on a guarded entry accept. Specifically, if a task containing a counter has become permanently blocked waiting for an entry to be called in a `select` statement, and if that entry is guarded by an expression

containing the counter variable, then the counter variable must have a value that allows the guard to be true. For example, if a task becomes permanently blocked waiting for a `LEAVE` entry call in the `select` statement of Figure 4.5, then the value of the variable `count` must be greater than zero. I can add an inequality to enforce this as follows. If x represents the transition variable for the arc with the `hang_a(leave)` symbol and c is a variable equated to the current value of `count`, then the inequality $c - x \geq 0$ will allow the arc for x to be traversed only if the counter is greater than zero. The condition $c > 0$ is said to be a *guard* for the transition labeled by x, extending the use of that term from the design language representation of a task to the automaton representation in a natural way. Note that, due to the constrained expression representation of CEDL, at most one `hang_a` symbol will occur for an entry, so $x \leq 1$. Even when the identical tasks technique described in the last section is used, this is still true since replicated tasks cannot contain `hang_a` symbols. This fact is crucial to the correctness of the inequality.

In general, given a counter variable whose value is held by an ILP variable c that can take on values between 0 and the constant n, I can prevent the transition with variable x from occurring except when a legal test on the counter is true at the end of the execution using one of the four inequalities shown in Table 4.1. This assumes that the transition could be taken at most once (i.e., $x \leq 1$). If a guard consists of several of these four legal tests conjoined together, then the inequalities for all of the tests are added to the system (since linear programming constraints are naturally conjunctive). There are techniques for handling disjunctive constraints in linear programming, but they may not be practical and I have not yet tried to apply them. Hence I do not allow the disjunction of tests on counter variables for this technique.

The second type of end condition involves a task that has terminated abnormally when a variable has gone out of range. Every state in an FSA that can accept an increment symbol for a counter variable can also accept a `use` symbol indicating that the counter is at its upper bound. This second alternative represents an overflow—the

```
select
  when count > 0 => accept LEAVE;
or
  accept ENTER;
end select;
```

Figure 4.5. Select Statement for End Condition Example

Table 4.1. Inequality Added to Enforce End Condition

Test	Inequality
$c = 0$	$c + nx \leq n$
$c > 0$	$c - x \geq 0$
$c = n$	$c - nx \geq 0$
$c < n$	$c + x \leq n$

erroneous attempt to increment the counter when it is at its upper bound—and leads to a dead-end accepting state. These alternatives are similar to those at each communication event that allow the task to permanently block. Decrement symbols have analogous alternatives, representing underflow, labeled with a use symbol indicating the counter is zero. Neither overflow nor underflow alternatives can be chosen unless the counter variable has an appropriate value, and I can add inequalities, like those described above, to enforce these end conditions. In the case where an increment or decrement is within the scope of a test of a counter variable (e.g., within an `if` or guarded entry of a `select`), this test must be satisfied if the overflow or underflow alternative is chosen. Additional inequalities are added to enforce these end conditions as well. For example, an `if` statement might specify that a counter c be decremented only if it is strictly positive. The underflow alternative for this decrement would then have the two end conditions $c = 0$ and $c > 0$ (note that these are inconsistent, thus the transition could never be taken).

I will illustrate this second type of end condition in the following example, to which I apply both the identical tasks technique of Section 4.1 and the counter variable

technique. The system consists of two resource allocators and R identical customers. Allocator $i = 1, 2$ initially has n_i units of its resource. Customers rendezvous with an allocator to acquire a unit of the allocator's resource or release a unit of the resource. Each customer repeats the following: acquire a unit of resource 1, acquire a unit of resource 2, release the unit of resource 2, and release the unit of resource 1. Each allocator maintains a count of the number of units of its resource currently available. This count is decremented whenever a customer acquires a unit and incremented whenever a customer releases a unit. The two resource allocators differ in only one way: allocator 1 will not accept requests for a resource when it is temporarily unable to grant the resource (the communication representing the granting of the resource unit is guarded by the counter representing the number of units of the resource remaining), while allocator 2 will abort if it ever receives a request it cannot fulfill. If $n_1 \leq n_2$, the second allocator will be protected by the guard on the first allocator and will never abort.

In Figure 4.6 I give a partial specification for this system in Ada showing one customer task and the two resource allocator tasks. As mentioned above, full automation of this technique will require extending CEDL and changing the deriver tool. This specification shows how counter variables could be presented to an augmented deriver tool in an extended CEDL.

Figure 4.7 shows a somewhat abbreviated set of FSAs for this system. The event symbols `a`i and `r`i represent the acquisition and release of a resource unit from allocator $i = 1, 2$. The event symbols `inc(c`i`)` and `dec(c`i`)` represent the increment and decrement operations on counter variable `c`i. I have annotated transition arcs having end conditions with these conditions in square brackets. These annotations should be regarded as the FSA equivalent of a comment—they are not actually present in the FSAs and do not enable/disable transitions. The end conditions they indicate will be enforced in the analysis using inequalities. Above each allocator FSA, I have placed an Ada-like declaration of its counter variable. For example, `c1:0..n1:=n1`

```
task body ALLOC1 is
C1 : INTEGER range 0..N1 := N1;
begin
  loop
    select
      when (C1 > 0) =>
        accept ACQUIRE1;
        C1 := C1 - 1;
    or
        accept RELEASE1;
        C1 := C1 + 1;
    end select;
  end loop;
end ALLOC1;

task body ALLOC2 is
C2 : INTEGER range 0..N2 := N2;
begin
  loop
    select
        accept ACQUIRE2;
        C2 := C2 - 1;
    or
        accept RELEASE2;
        C2 := C2 + 1;
    end select;
  end loop;
end ALLOC2;

task body CUSTOMER is
begin
  loop
    ALLOC1.ACQUIRE1;
    ALLOC2.ACQUIRE2;
    ALLOC2.RELEASE2;
    ALLOC1.RELEASE1;
  end loop;
end CUSTOMER;
```

Figure 4.6. Ada for Coupled Resource Allocator Example

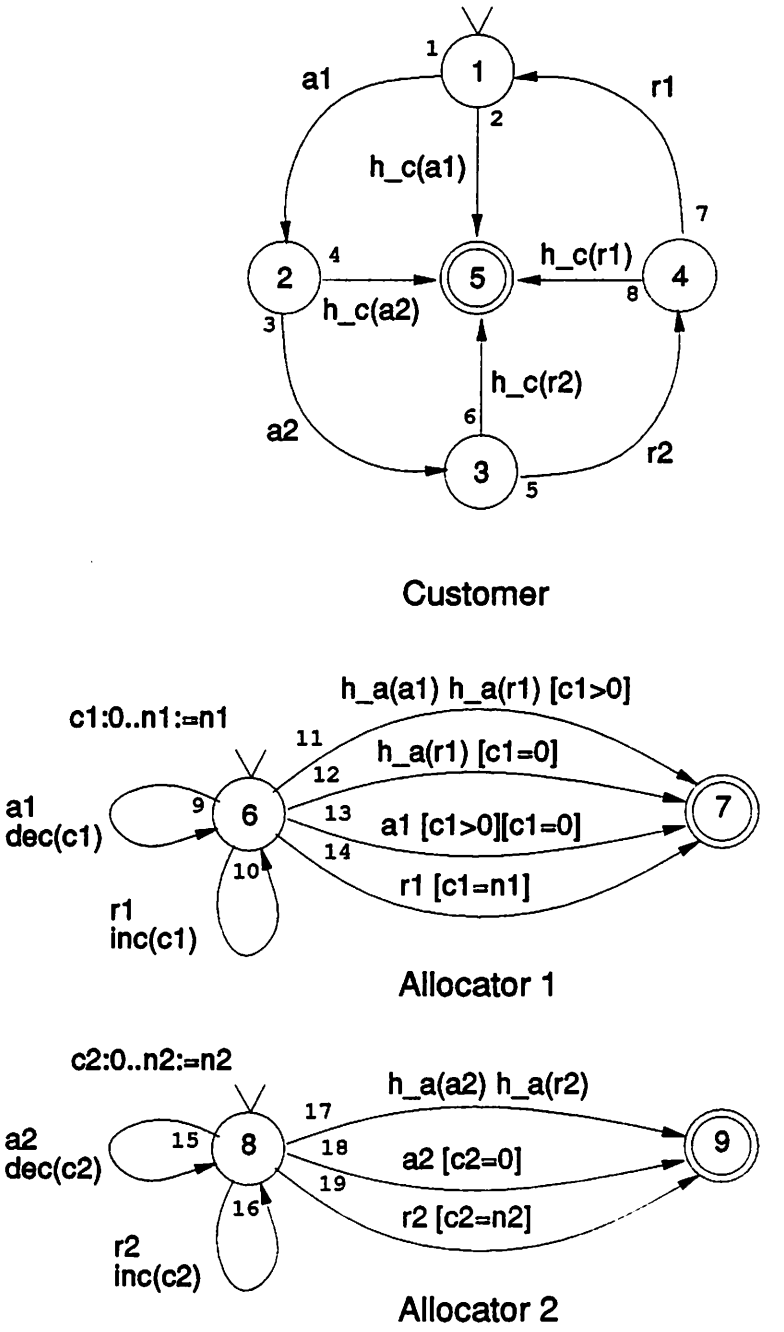


Figure 4.7. FSAs for Coupled Resource Allocator Example

declares the counter c1 to have range 0..n1 and have initial value n1. Transitions 13 and 14 represent the underflow and overflow of counter 1 respectively, and transitions 18 and 19 represent these alternatives for counter 2. Note that since the end conditions for the underflow alternative in allocator 1 are inconsistent, that transition can never be taken (the end condition inequalities force $x_{13} = 0$). Transitions 11 and 12 represent the hang alternatives for allocator 1. Since this allocator will not communicate on channel a1 if c1=0, end conditions allow the allocator to permanently block waiting for this communication only if c1>0. Transition 17 represents the hang alternative for allocator 2. The inequality system generated from these FSAs is shown in Figure 4.8. The additional ILP variables c_1, c_2 are introduced to hold the values of the resource unit counters for allocators 1 and 2. The *counter equations* set these variables to their values at the end of the trace, the *range inequalities* enforce the upper bounds on the counters (since the ILP variables are nonnegative, the lower bound of zero is enforced implicitly), and the *end condition inequalities* use these values to constrain which task termination alternatives can be chosen.

Depending on the values of the constants R, n_1, and n_2, the system may or may not have a finite trace. Since the tasks involved loop forever, the only finite traces are those in which the tasks become permanently blocked or abort, as would happen, for example, if the second resource allocator received more requests than it could accommodate and aborted, causing the remaining tasks to become permanently blocked. This may occur only if $R > n_2$ and $n_1 > n_2$. If either of these conditions is not met, then the inequality system has no integral solution, proving that no finite traces exist in this case. On the other hand, if $R > n_2$ and $n_1 > n_2$, then there is an integer solution to the inequality system. Suppose $R = n_1 = 10$ and $n_2 = 5$. Then there is a solution where $x_4 = 4$, $x_6 = 6$, and $x_{12} = x_{18} = 1$ (only the nonzero termination alternatives for each task have been listed) corresponding to a trace in which ten customers get a unit of resource 1, five customers get a unit of resource 2, and the sixth customer's request causes resource allocator 2 to abort. The six

Flow:			**(state)**
$R + x_7$	$=$	$x_1 + x_2$	(1)
x_1	$=$	$x_3 + x_4$	(2)
x_3	$=$	$x_5 + x_6$	(3)
x_5	$=$	$x_7 + x_8$	(4)
$x_2 + x_4 + x_6 + x_8$	$=$	f_5	(5)
$1 + x_9 + x_{10}$	$=$	$x_9 + x_{10} + x_{11} + x_{12} + x_{13} + x_{14}$	(6)
$x_{11} + x_{12} + x_{13} + x_{14}$	$=$	f_7	(7)
$1 + x_{15} + x_{16}$	$=$	$x_{15} + x_{16} + x_{17} + x_{18} + x_{19}$	(8)
$x_{17} + x_{18} + x_{19}$	$=$	f_9	(9)
Synchronization:			**(channel)**
x_1	$=$	$x_9 + x_{13}$	$(a1)$
x_3	$=$	$x_{15} + x_{18}$	$(a2)$
x_7	$=$	$x_{10} + x_{14}$	$(r1)$
x_5	$=$	$x_{16} + x_{19}$	$(r2)$
Hang:			**(channel)**
$x_2 + Rx_{11}$	$\leq$	R	$(a1)$
$x_4 + Rx_{17}$	$\leq$	R	$(a2)$
$x_8 + Rx_{11} + Rx_{12}$	$\leq$	R	$(r1)$
$x_6 + Rx_{17}$	$\leq$	R	$(r2)$
Counter:			**(counter)**
c_1	$=$	$n_1 + x_{10} - x_9$	(1)
c_2	$=$	$n_2 + x_{16} - x_{15}$	(2)
Range:			**(counter)**
c_1	$\leq$	n_1	(1)
c_2	$\leq$	n_2	(2)
End Condition:			**(arc)**
$c_1 - x_{11}$	$\geq$	0	(11)
$c_1 + n_1 x_{12}$	$\leq$	n_1	(12)
$c_1 - x_{13}$	$\geq$	0	(13)
$c_1 + n_1 x_{13}$	$\leq$	n_1	(13)
$c_1 - n_1 x_{14}$	$\geq$	0	(14)
$c_2 + n_2 x_{18}$	$\leq$	n_2	(18)
$c_2 - n_2 x_{19}$	$\geq$	0	(19)

Figure 4.8. Inequality System for Coupled Resource Allocator Example

customers that completed a rendezvous with allocator 2 block waiting to release the resource, while the other four block waiting to acquire it. For simplicity, I have not modeled the refusal of allocator 2 to grant a resource unit; the sixth a2 communication completes before the allocator aborts, allowing the sixth customer to think it has a unit of the resource.

The algorithm for generating the additional inequalities used by this technique is shown in Figure 4.9. For a task variable v with range $0, \ldots, n_v$ used as a counter, v_0 is the initial value of v, *inc(v)* is the set of transitions labeled with increments of v, *dec(v)* is the set of transitions labeled with decrements of v, *over(v)* is the set of transitions labeled with **use** symbols representing an increment of the counter when at its upper limit (an overflow), and *under(v)* is the set of transitions labeled with **use** symbols representing a decrement of the counter when zero (an underflow).

4.2.1 Comparison with Constraint Elimination

The technique described above is intended to replace constraint elimination in some cases as a means for dealing with dataflow information in the analysis. The technique is not as powerful as constraint elimination since it cannot remove all spurious solutions that are due to violation of dataflow constraints (as constraint elimination can, at least in principle). There are two reasons for this. First, the technique can only be applied to variables used as counters in the manner described above. If arbitrary assignments are made to variables, the value of the variable at the end of the trace clearly depends on the order in which these assignments are made. If the assignments are made within a loop, the inequality system does not represent the order in which they are made. For example, suppose arcs 9 and 10 were labeled with assignments and were each traversed once in the allocator FSA of Figure 4.7. Then $x_9 = x_{10} = 1$, but this says nothing about the order in which the arcs were traversed. Thus it seems unlikely that this technique will ever generalize for use on variables not used as counters.

Input: A set M of FSAs
A set of task variables V used as counters
Output: A set of inequalities enforcing end conditions

For each task variable v in V:
Create counter variable c_v
Generate counter equation: $c_v = v_0 + \sum_{k \in inc(v)} x_k - \sum_{k \in dec(v)} x_k$
Generate range inequality: $c_v \leq n_v$
For each channel c:
For each transition $k \in hang\_a(c)$:
For each test t of a counter variable v in the guard for k:
Case t of
$v = 0$: Generate end condition inequality: $c_v + n_v x_k \leq n_v$
$v > 0$: Generate end condition inequality: $c_v - x_k \geq 0$
$v = n_v$: Generate end condition inequality: $c_v - n_v x_k \geq 0$
$v < n_v$: Generate end condition inequality: $c_v + x_k \leq n_v$
For each task variable v in V:
For each transition $k \in over(v)$:
Generate end condition inequality: $c_v - n_v x_k \geq 0$
For each transition $k \in under(v)$:
Generate end condition inequality: $c_v + n_v x_k \leq n_v$
For each transition $k \in over(v) \cup under(v)$:
For each test t of a counter variable v that k is in the scope of:
Case t of
$v = 0$: Generate end condition inequality: $c_v + n_v x_k \leq n_v$
$v > 0$: Generate end condition inequality: $c_v - x_k \geq 0$
$v = n_v$: Generate end condition inequality: $c_v - n_v x_k \geq 0$
$v < n_v$: Generate end condition inequality: $c_v + x_k \leq n_v$

Figure 4.9. Algorithm for Counter Variables

Even if variables are used as counters, the technique is still not as powerful as constraint elimination since it only checks restrictions on the use of a variable at the end of a trace. This allows the analysis to select paths through an FSA that violate a dataflow constraint without violating any end conditions. Such a path represents an impossible behavior of a task in which a branch is made assuming a variable has a value different than the one it actually has, but either:

- the manner in which the task terminates does not produce end conditions on that variable, or
- by the end of the trace, the variable has been given a value satisfying all the end conditions.

For each of these two cases, I show an example of a system for which a spurious deadlock is reported using the counter variable technique that would not be reported if constraint elimination had been used to account for dataflow.

First, let us examine the case where the manner in which a task terminates does not generate end conditions for the counter variable whose dataflow constraint is violated. Consider the concurrent system in Figure 4.10. M_1 makes a nondeterministic choice and signals this choice to M_2 via the rendezvous a or b. M_2 records this choice in the counter variable **count** by incrementing the variable only if M_1 chose a. M_2 then branches on the value of the counter and acknowledges the choice made by responding with c for a or with d for b. Without enforcing the dataflow constraint for the counter, the analysis can select a path through M_2 violating this constraint and report a spurious deadlock in which M_2 becomes permanently blocked making the wrong response. The technique described in this section does not suffice to remove this spurious deadlock from the analysis. Since the termination alternatives represent unguarded entry accepts and calls, there are no end conditions to be enforced. As a result, there exist solutions to the inequality system in which the path through M_2 violates the dataflow constraint for the counter.

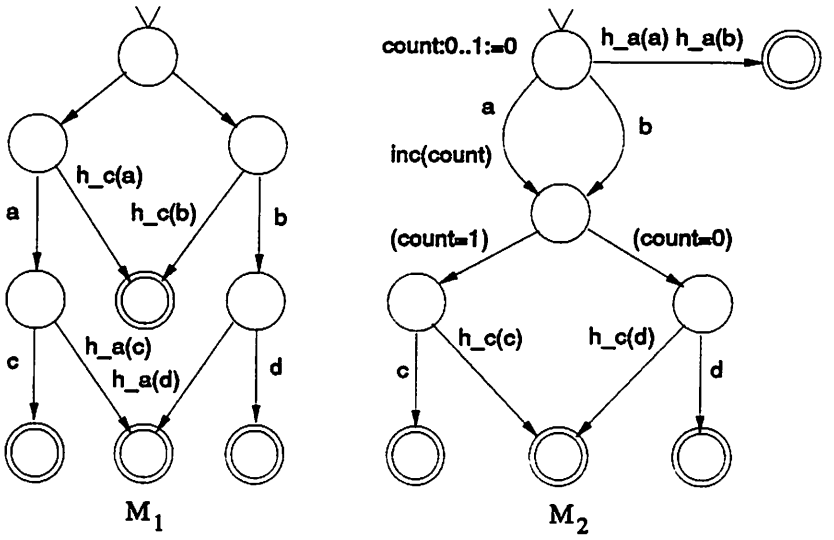


Figure 4.10. Example Where Counter Variable Technique Fails

Now consider the case where a counter variable is restored to a value satisfying the end condition before the end of the trace. Here, we return to the possibility, mentioned earlier, that the order of increments and decrements on a counter sends the counter out of its range. Since the order of the increments and decrements is ignored in the calculation of the final counter value, the analysis admits solutions corresponding to "traces" where the counter is temporarily out of its range. For example, consider a task that initializes a counter to zero, decrements its counter, then increments its counter. Such a task should be forced to abort on the decrement—constraint elimination would ensure this—but since the counter is restored to a legal value before the task terminates, no end condition could force the correct behavior.

I argue that this limitation is usually not relevant in the context of this analysis technique. I believe that counters will typically be changed in response to communications with other tasks, as in the example of Figure 4.7, and since the analysis technique itself does not represent the order in which these communications take place (only the total number of each), no additional spurious solutions are introduced by not representing the order of the counter variable changes that are dependent on the order

of these communications. For example, suppose the customer task in Figure 4.7 were changed to release the resource, acquire it, and then terminate (say $R = n_1 = n_2 = 1$). Then even if the resource allocators underwent constraint elimination, the analysis would still find a spurious solution corresponding to a "trace" where the customer terminates normally, even though this must cause the counters to overflow. This is not a problem with dataflow, but with the order of the communications between the tasks not being enforced by the inequalities representing the synchronization constraints, as discussed in Section 3.2.

4.3 Summary

Frequently, compactly specified concurrent systems have a large number of states because they contain many identical tasks and/or because they have many variables or variables with large ranges. In this chapter, I have presented two techniques for efficiently analyzing these types of concurrent systems. In the first technique, I showed that finding a flow of R through a task's FSA effectively creates R copies of the task in the context of the analysis. This technique can be used to create large numbers of identical tasks in a concurrent system without increasing the size of the inequality system used in the analysis. The tasks created are identical in the sense that any other task must be willing to communicate with all the tasks if it is willing to communicate with any one (i.e., other tasks cannot distinguish between the identical tasks with communication). In the second technique, I use an ILP variable to represent the value of a program variable used as a counter. This avoids the use of constraint elimination for that variable, which can cause a state explosion in the task's FSA if the task has many variables or if the variables have large ranges. The technique only applies when the program variable is used as a counter, however, and even then the necessary conditions generated are not as strong as those generated if constraint elimination were used. A limited empirical evaluation of these techniques is presented in Section 7.2.1.

CHAPTER 5

VERIFYING MORE COMPLEX PROPERTIES

This chapter describes extensions to the constrained expression analysis technique for concurrent systems that allow more complex concurrency properties to be verified. Section 5.1 describes a technique for determining if a pattern of event symbols, specified by a restricted kind of regular expression, can occur in any trace. Section 5.2 extends the formalism and analysis technique to infinite executions. In Section 5.3, I combine the techniques of Sections 5.1 and 5.2 into a technique for determining if a pattern of events specified by an ω-regular expression of a restricted form can occur in any infinite execution of the system. Section 5.4 describes an even more powerful technique that can determine whether a sequence of events specified by an FSA or a Büchi automaton can occur in any trace. Finally, Section 5.5 describes a technique for detecting critical races.

5.1 Patterns of Events

The constrained expression analysis technique, presented in Section 3.2, allows only queries for traces that can be characterized by linear relations over the counts of specific event symbols. In this section, I present a technique for extending this analysis to queries involving patterns of events specified by *star-less*[1] *expressions*, which are regular expressions formed from event symbols, plus the concatenation and union operators (but not Kleene star). Such expressions can always be reduced to disjunctive normal form. Section 5.1.1 discusses how to answer queries about a single sequence of events. Section 5.1.2 shows how to combine the analyses for multiple

[1]Not to be confused with *star-free* expressions, which also allow the negation operator.

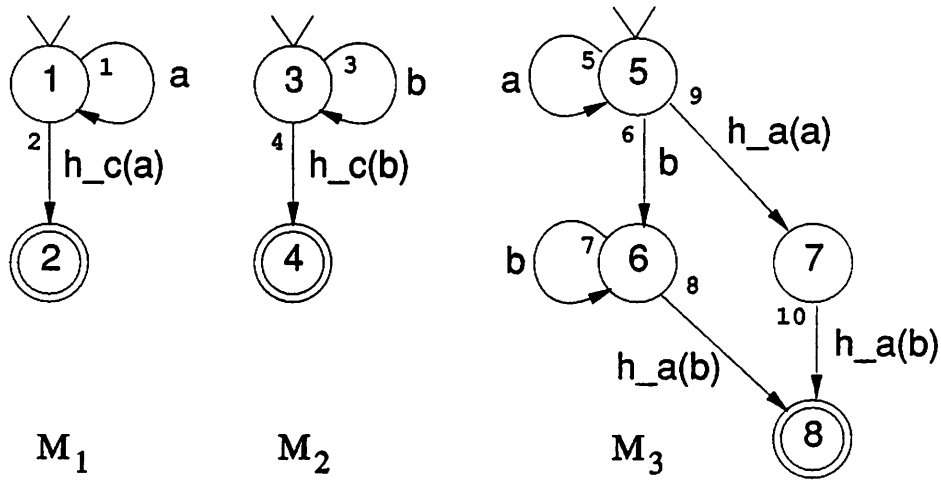


Figure 5.1. Example for Sequence Query

sequences of events into one analysis for a disjunction of these sequences. The results of experiments with this technique and another technique to represent infinite traces described in Section 5.2, are presented together in Section 7.2.2.

5.1.1 Sequence of Events

The original constrained expression analysis technique can easily find traces in which certain event symbols occur a specified number of times, but it cannot find traces in which these symbols occur in a specific order. For example, to find a trace with one a event and one b event in the system of Figure 5.1, $x_1 = 1$ and $x_3 = 1$ would be added to the inequality system for this example, which is shown in Figure 5.2. There does not appear to be any way, however, to add equations requiring that the events occur in a specific order. This section describes a technique for deriving necessary conditions for the existence of a trace (or prefix of a trace) in which a specific sequence of events occurs.

Given a sequence of events $e_1 e_2 \ldots e_n$ as a query, where $\Pi = \bigcup_{i=1}^{n} \{e_i\}$ is the alphabet of the query, I construct an inequality system representing necessary conditions for the existence of a trace (or a prefix of a trace) t such that $\rho_\Pi(t) = e_1 e_2 \ldots e_n$ (i.e., the projection of the trace on the alphabet of the query is equal to the sequence).

$$
\begin{array}{rcll}
\textbf{Flow:} & & & \textbf{(state)} \\
1 + x_1 & = & x_1 + x_2 & (1) \\
x_2 & = & f_2 & (2) \\
1 + x_3 & = & x_3 + x_4 & (3) \\
x_4 & = & f_4 & (4) \\
1 + x_5 & = & x_5 + x_6 + x_9 & (5) \\
x_6 + x_7 & = & x_7 + x_8 & (6) \\
x_9 & = & x_{10} & (7) \\
x_8 + x_{10} & = & f_8 & (8) \\
\textbf{Synchronization:} & & & \textbf{(channel)} \\
x_1 & = & x_5 & (a) \\
x_3 & = & x_6 + x_7 & (b) \\
\textbf{Hang:} & & & \textbf{(channel)} \\
x_2 + x_9 & \leq & 1 & (a) \\
x_4 + x_8 + x_{10} & \leq & 1 & (b)
\end{array}
$$

Figure 5.2. Inequality System to Find Trace

To accomplish this, I conceptually divide the trace into *intervals*, generate a different inequality system for each interval, and connect these inequality systems together.

An interval is a segment of a trace having certain characteristics, including restrictions on the events it can contain and where it can begin and end. For a sequence query with n event symbols, I divide the trace into $n+1$ intervals. The first interval is from the start of the trace up to e_1. The i^{th} interval ends with e_i, and the last interval is the remainder of the trace after e_n. If only a prefix of a trace is sought, then the last interval is unnecessary. For each interval, I generate an inequality system, which I call an *interval system*, similar to the one normally generated in a constrained expression analysis. Just as the normal inequality system "finds" the actions of the FSAs for the whole trace, each interval system "finds" the actions of the FSAs for the part of the trace in its interval. Each interval system has flow and synchronization equations in its own transition variables. An interval system differs from the inequality system normally generated in three ways:

1. No hang inequalities are generated for an interval system. Instead, one set of hang inequalities is shared by all the interval systems. These inequalities

require that the sum of transition variables for arcs labeled with hang_c or hang_a symbols for a given entry call over all the interval systems must be less than or equal to one. This ensures that both a hang_c and a hang_a symbol for the same channel do not appear in a trace, despite its being divided into intervals.

2. Additional inequalities, called *requirement inequalities*, are added to interval system i ($i = 1, \ldots, n$) to insure that the projection of the interval on Π will be e_i (i.e., no symbols from Π may appear except e_i, which appears once). The inequalities for interval i simply set the number of occurrences of the events e_j for $j \neq i$ equal to zero, and set the number of occurrences of event e_i to one.

3. Rather than finding a flow in each FSA from a starting to an accepting state, an interval system finds a flow in each FSA from some state the FSA could be in at the start of the interval to some state the FSA could be in at the end of the interval.

I first discuss how the last item is accomplished, and then illustrate the technique with an example.

For the first interval, the FSAs must be in their start states at the beginning of the interval. Similarly, the FSAs must be in accepting states at the end of interval $n + 1$ (if present). From the constrained expression technique, I know how to force the flow to begin at start states and end at accepting states. Instead, I want to force flow to begin or end after certain event symbols (i.e., the e_i). Except for interval 1, the interval systems do not add an implicit flow in of one to the start states of the FSAs. Also, except for interval $n + 1$ (if present), the interval systems do not contain accept variables f_j counted as flow out of FSA accepting states. Instead, the interval systems make use of a new kind of variable to force flows to begin and end at specific states in the FSAs. These variables are assigned to states in which an FSA could be at the beginning or end of an interval. These states are determined as follows. Immediately after an e_i event, all FSAs containing event e_i must be in a

state with an incoming e_i transition and all other FSAs could be in any state (as far as we can tell without further analysis). Therefore, to each state j in which an FSA could be following the e_i event ending interval i, I assign a *connection variable*, $c_{i,j}$, representing the number of times this FSA ends interval i in state j (this will be one if the FSA is in state j at the end of the interval, and zero otherwise). This variable is counted as flow out of state j in interval i and is counted as flow into state j in interval $i+1$ (if present). If interval $n+1$ is not present, then the $c_{n,j}$ variables act like the accept variables in the inequality system generated by the original technique, allowing the flow to drain from the FSA.

These variables connect the interval systems in such a way that a flow will be found in each FSA. This flow will proceed as follows: it will start in the starting state of the FSA, proceed through interval 1 (i.e., through transition variables in interval system 1) to a state in which the FSA could be after an e_1 event, switch from interval 1 to interval 2 via some connection variable, proceed through interval 2 to a state in which the FSA could be after an e_2 event, and continue in this manner to an accepting state in interval $n+1$, if present. If interval $n+1$ is not present, the flow will end at a state in which the FSA could be following an e_n event in interval n.

I illustrate the technique using the example of Figure 5.1. I want to determine if there exists a prefix of a trace of this system matching the star-less expression ba (i.e., a prefix t such that $\rho_{\{a,b\}}(t) = ba$). I divide the trace into two intervals. The first interval is from the initial state of the system to the state of the system after the b event. The second interval is from the state of the system after the b event to the state of the system after the a event. I want the first interval system to find flows ending after a b event, so I assign a connection variable $c_{1,j}$ to each state j having an incoming b transition in M_2 and M_3, and I assign a connection variable $c_{1,j}$ to every state j of M_1, which does not contain b events. Similarly, in the second interval, I assign a connection variable $c_{2,j}$ to each state j having an incoming a transition in M_1 and M_3, and I assign a connection variable $c_{2,j}$ to every state j of M_2, which does

not contain a events. I add requirement inequalities to force the number of b events in the first interval to one, the number of a events in the second interval to one, the number of a events in the first interval to zero, and the number of b events in the second interval to zero.

The inequality system generated for this example is shown in Figure 5.3. The transition variable for transition k of interval i is denoted $x_{i,k}$. The whole system finds a flow in each FSA starting at the start state, proceeding through the first interval to a state with a connection variable for b, and then continuing through the second interval to a state with a connection variable for a.

Note that this inequality system, which represents necessary conditions for a trace containing a b and then an a to exist, has no integral solution. This proves that no trace matching the query ba exists. For this trivial example, an appropriate kind of intersection between M_3 and the automaton for ba could have shown this. The above technique, however, will work even if the events a and b are in different FSAs.

The algorithm for generating inequalities that represent necessary conditions for the existence of a trace t such that $\rho_\Pi(t) = e_1 e_2 \ldots e_n$ is shown in Figure 5.4. For an event symbol e, *occur(e)* is the set of transitions labeled with e from any one FSA containing e. This set can be used to compute the number of occurrences of e in interval i:

$$\sum_{k \in occur(e)} x_{i,k}$$

The function *label(k)* returns the label of transition k. The set *alphabet(M_i)* is the alphabet of FSA M_i. To generate the inequalities that represent necessary conditions for the existence of a prefix of a trace, the variables and inequalities for interval $n+1$ are not generated (simply let i range over $1, \ldots, n$ instead of $1, \ldots, n+1$ in the `for` loops and in the summation in the hang inequality).

If a communication between two tasks is represented by pairs of distinct event symbols in their respective FSAs, as described in Section 3.2, then I make the following restriction on the query for technical reasons: at most one of the symbols used to

Flow (Interval 1):			**(state)**
$1 + x_{1,1}$	$=$	$x_{1,1} + x_{1,2} + c_{1,1}$	(1)
$x_{1,2}$	$=$	$c_{1,2}$	(2)
$1 + x_{1,3}$	$=$	$x_{1,3} + x_{1,4} + c_{1,3}$	(3)
$x_{1,4}$	$=$	0	(4)
$1 + x_{1,5}$	$=$	$x_{1,5} + x_{1,6} + x_{1,9}$	(5)
$x_{1,6} + x_{1,7}$	$=$	$x_{1,7} + x_{1,8} + c_{1,6}$	(6)
$x_{1,9}$	$=$	$x_{1,10}$	(7)
$x_{1,8} + x_{1,10}$	$=$	0	(8)
Synchronization (Interval 1):			**(channel)**
$x_{1,1}$	$=$	$x_{1,5}$	(a)
$x_{1,3}$	$=$	$x_{1,6} + x_{1,7}$	(b)
Requirement (Interval 1):			**(symbol)**
$x_{1,1}$	$=$	0	(a)
$x_{1,3}$	$=$	1	(b)
Flow (Interval 2):			**(state)**
$c_{1,1} + x_{2,1}$	$=$	$x_{2,1} + x_{2,2} + c_{2,1}$	(1)
$c_{1,2} + x_{2,2}$	$=$	0	(2)
$c_{1,3} + x_{2,3}$	$=$	$x_{2,3} + x_{2,4} + c_{2,3}$	(3)
$x_{2,4}$	$=$	$c_{2,4}$	(4)
$x_{2,5}$	$=$	$x_{2,5} + x_{2,6} + x_{2,9} + c_{2,5}$	(5)
$c_{1,6} + x_{2,6} + x_{2,7}$	$=$	$x_{2,7} + x_{2,8}$	(6)
$x_{2,9}$	$=$	$x_{2,10}$	(7)
$x_{2,8} + x_{2,10}$	$=$	0	(8)
Synchronization (Interval 2):			**(channel)**
$x_{2,1}$	$=$	$x_{2,5}$	(a)
$x_{2,3}$	$=$	$x_{2,6} + x_{2,7}$	(b)
Requirement (Interval 2):			**(symbol)**
$x_{2,1}$	$=$	1	(a)
$x_{2,3}$	$=$	0	(b)
Hang:			**(channel)**
$x_{1,2} + x_{1,9} + x_{2,2} + x_{2,9}$	$\leq$	1	(a)
$x_{1,4} + x_{1,8} + x_{1,10} + x_{2,4} + x_{2,8} + x_{2,10}$	$\leq$	1	(b)

Figure 5.3. Inequality System for Query *ba*

Input: A set M of FSAs
A sequence of events $e_1 e_2 \ldots e_n$
Output: A set of inequalities

For each interval $i = 1, \ldots, n+1$:
  For each transition k in an FSA of M:
    Create transition variable $x_{i,k}$
  If $i \neq n+1$ then
    For each state j in an FSA of M:
      If $\exists k \in in(j)(label(k) = e_i \vee e_i \notin alphabet(fsa(j)))$ then
        Create connection variable $c_{i,j}$
For each accepting state j of an FSA of M:
  Create accept variable f_j
For each interval $i = 1, \ldots, n+1$:
  For each state j of an FSA of M:
    Generate flow equation:

$$[1]_{i=1 \wedge start(j)} + [c_{i-1,j}] + \sum_{k \in in(j)} x_{i,k} = \sum_{k \in out(j)} x_{i,k} + [c_{i,j}] + [f_j]_{i=n+1}$$

For each interval $i = 1, \ldots, n+1$:
  For each channel c:
    Generate synchronization equation: $\sum_{k \in call(c)} x_{i,k} = \sum_{k \in accept(c)} x_{i,k}$
  For each e in $\{e_1, \ldots, e_n\}$:
    If $e = e_i$ then
      Generate requirement equation: $\sum_{k \in occur(e)} x_{i,k} = 1$
    Else
      Generate requirement equation: $\sum_{k \in occur(e)} x_{i,k} = 0$
For each channel c:
  Generate hang inequality:

$$\sum_{i=1}^{n+1} \left(\sum_{k \in hang\_c(c)} x_{i,k} + \sum_{k \in hang\_a(c)} x_{i,k} \right) \leq 1$$

Figure 5.4. Algorithm for Sequence

represent a given communication may appear in the sequence $e_1 e_2 \ldots e_n$. Though their symbols are ordered in the trace by the total order semantics of the formalism, the events representing a synchronization conceptually happen at the same time, as enforced by the synchronization equations in the analysis technique. This restriction prevents anomalous sequences such as "`call(c) beg_rend(c)`" which the technique would conclude does not match any trace since the matching communication events are forced to occur in different intervals.

Theorem 5.1 *The inequalities generated by the algorithm in Figure 5.4 are necessary conditions for the existence of a trace t of the system such that $\rho_\Pi(t) = e_1 e_2 \ldots e_n$.*

Proof. Suppose there exists a trace t such that $\rho_\Pi(t) = e_1 e_2 \ldots e_n$. I describe a solution to the inequality system generated by the algorithm in Figure 5.4 corresponding to this trace. The (augmented) trace t describes a path through each FSA from a starting state to an accepting state. Each such path can be divided into $n+1$ segments delimited by the e_i events: segment $i = 1, \ldots, n$ ends in the state the FSA was in immediately following event e_i. Set $x_{i,k}$ equal to the number of times transition arc k is crossed by a path in segment i. Set $c_{i,j}$ to one if the FSA containing state j is in state j immediately after the occurrence of event e_i, and otherwise set it to zero. Set f_j to one if the FSA containing state j is in state j at the end of the trace, and otherwise set it to zero. Since the connection variables can be seen to simply transfer the flow in each FSA from one interval to another, the above solution satisfies the flow equations. Matching synchronization events occur in the same interval, thus the synchronization equations equating their totals are satisfied. Since the projection of interval i on $\{e_1, \ldots, e_n\}$ is e_i, the requirement equations are satisfied. Finally, since at most one task can be blocked waiting for communication on a particular channel, the hang inequalities are satisfied. □

The size of the inequality system generated for the sequence $e_1 \ldots e_n$ by the algorithm in Figure 5.4 is roughly $n + 1$ times the size of the inequality system

generated by the algorithm for the original technique in Figure 3.2. If only a prefix of a trace is sought, then the inequality system for the sequence is roughly n times the size of the inequality system generated for the original technique.

For an asynchronous communication mechanism, the technique must be adjusted slightly. Messages sent in an interval may be received in that interval or in any subsequent interval. Therefore, when generating the communication inequalities for an interval, the totals for the number of messages sent and the number of messages received should include send and receive events from the current interval plus all preceding intervals. For example, the communication inequality for interval 2 would restrict the number of message receptions for a given channel in intervals 1 and 2 to be less than or equal to the number of message sends to that same channel for intervals 1 and 2.

I conclude this section by mentioning a few minor extensions to the technique. First, I can use requirement inequalities to require or forbid other events from occurring in intervals, in addition to those named in the query. For example, I might formulate a query about mutual exclusion as follows: "Does there exist a trace in which customer 1 begins to use the resource and then customer 2 begins to use the resource before customer 1 finishes using the resource?" Such a query could be represented with two intervals: the first ending with customer 1 beginning to use the resource, and the second ending with customer 2 beginning to use the resource and not allowing an occurrence of the event representing customer 1 finishing with the resource. The requirement equation to forbid an event simply sets the number of occurrences of that event to zero, while the requirement inequality to require an event sets the number of occurrences of that event to be greater than or equal to one. In addition, I permit arbitrary linear relations to be added to each interval system (e.g., the total number of a's and b's in the interval is five), just as such relations could be added in the original analysis technique. Finally, I note that an interval need not end with a specific event. I might, for example, simply create an interval in which an

event is required. Without an ending event, however, connection variables must be assigned to every state.

The second minor extension allows the generation of necessary conditions for the existence of a trace in which a sequence of event symbols matching a query occurs after some of the events in the query alphabet have occurred in an order not matching the query. In other words, I can generate necessary conditions for the existence of a trace t such that $\rho_\Pi(t) = \Pi^* e_1 e_2 \ldots e_n$. I can accomplish this by simply relaxing the requirement inequalities in the first interval. The equations setting the number of occurrences of e_1 to one are changed to inequalities requiring at least one e_1 occur, and the other requirement inequalities, which forbid the other events in the query alphabet, are removed. The placement of the connection variables will guarantee that the first interval still ends with an e_1 event. I call an interval in which the requirement inequalities have been so relaxed an *open interval.* The first interval in a query is usually open since this seems like the most natural interpretation of most queries (i.e., can this sequence eventually occur). The analyst may also specify other intervals as open, as explained in Section 5.3.

5.1.2 Disjunctions of Sequences

In the last section, I showed how to generate necessary conditions for the existence of a trace matching a query consisting of a sequence of events. In this section, I extend this technique to generate necessary conditions for the existence of a trace matching a star-less query. Since all star-less queries can be put in disjunctive normal form, all that remains is to show how to generate necessary conditions for the existence of a trace in which one of a finite set of sequences occurs. Given a star-less query

$$\bigcup_{i=1}^{m} e_{i,1} e_{i,2} \ldots e_{i,n_i}$$

where $\Pi = \bigcup_{i,j} \{e_{i,j}\}$ is the alphabet of the query, I construct an inequality system representing necessary conditions for the existence of a trace t such that $\rho_\Pi(t) =$

$e_{i,1}e_{i,2}\ldots e_{i,n_i}$ for some $i = 1,\ldots,m$ (i.e., the projection of the trace on the alphabet of the query is equal to one of the sequences). I accomplish this by generating an inequality system for each sequence, as described in the last section, and then connecting them together.

Given a set of sequences, I derive necessary conditions for the existence of a trace matching the query consisting of their disjunction as follows. I assign a *sequence variable*, s_i, to each sequence that will be one if that sequence is the one found and zero otherwise. I generate an equation summing the sequence variables to one, forcing one sequence to be sought. I generate an inequality system, which I call a *sequence system*, for each sequence. A sequence system is similar to the inequality system generated for a sequence query, as described in Section 5.1.1, with the following exceptions:

- In the sequence system for sequence i, the implicit flow in of one at the start states in the first interval is changed to an implicit flow in of s_i. Thus flows will only be found in the sequence system for the sequence being sought.

- In the sequence system for sequence i, the requirement equations that require a symbol occur exactly (or at least) once are changed to require the occurrence of that symbol exactly (or at least) s_i times. Thus an event is not forced to occur unless the sequence in which it is contained is the one being sought.

If the sequence variable for a sequence system is set to one, it is the same as the inequality system that would be generated for the sequence standing alone. If the sequence variable for a sequence system is set to zero, it will always have the trivial solution where all the variables are zero. By this construction, it is clear that there exists an integral solution to the inequality system for the disjunction if and only if there exists an integral solution to at least one of the inequality systems for a sequence. Thus the resulting inequality system represents necessary conditions for the existence of a trace matching one of the sequences.

The size of the inequality system for the disjunction is equal to the sum of the sizes of the inequality systems for the sequences, plus one additional variable per sequence, plus one additional equation (summing the sequence variables to one). An optimization described in Section 7.1.3 can significantly reduce the size of the inequality system for a disjunction by having different sequences share the same transition variables.

In Section 5.3, I will combine the technique presented here with the technique for infinite traces described in Section 5.2. The results of experiments with this combined technique will be presented in Section 7.2.2.

5.2 Infinite Traces

Another limitation of the constrained expression analysis technique presented in Section 3.2 is that it does not admit infinite traces, i.e., traces in which one or more FSAs continue engaging in actions forever. Note that the inequality system in Figure 5.2 (on page 73) has no integral solution since all traces of the concurrent system in Figure 5.1 are infinite (there is no way for all of the FSAs to reach accepting states without violating the constraints involving the hang symbols). In fact, the constrained expression formalism described in Section 3.1 cannot even represent infinite traces since the formal languages used to model the system are over finite strings. To test for liveness properties, infinite traces must be represented, since the negation of a liveness property will be an expression forbidding some good event(s) from occurring in a potentially infinite execution. In this section, I show how the formalism can be extended to represent infinite traces and I describe a technique for dealing with infinite traces in the analysis.

The constrained expression formalism represents the behavior of a concurrent system with a finite string. To represent infinite traces, I instead represent the behavior of the system with an infinite string of event symbols and make two changes to the constrained expression representation of a system. First, the tasks of the

system are modeled by Büchi automata [73] rather than FSAs. A Büchi automaton is the infinite analog of an FSA, accepting languages of infinite strings rather than finite strings. The only difference is the condition for acceptance. In an FSA, a computation must end in an accepting state for the string to be accepted. A Büchi automaton accepts an infinite string if and only if the infinite computation on that string enters at least one of the accepting states of the automaton infinitely often. Büchi automata accept ω-regular languages, the infinite analog of regular languages. The Büchi automaton, M_ω, used to model a task looks exactly like the FSA, M, used before except that:

- All states in M_ω are accepting. This admits any infinite path through the automaton as a legal trace of the task.
- To each state of M_ω that was accepting in M, I add a self-loop (i.e., a transition from the state to itself) on a *stopped symbol*, denoted s_M, unique to the automaton. This allows legal finite behaviors of tasks within the framework of infinite strings: for every finite string t accepted by M there is an infinite string $t(s_M)^\omega$ accepted by M_ω.

The second change made to the formalism involves the constraints. Each constraint language is defined to be the union of a recursive language of finite strings over Σ and an ω-context-free language [73] of infinite strings over Σ. Constraint languages must include both finite and infinite strings since the projection of an infinite string onto an alphabet may be either finite or infinite. This completes the extension of the formalism. The analysis technique will still use the FSA representation of the tasks, but will treat them as Büchi automata where appropriate. As with the original analysis technique, I have no procedure to generate inequalities for arbitrary constraint languages, but I know how to generate inequalities for constraint languages that enforce common communication mechanisms (e.g., synchronous and asynchronous message passing).

I now describe a technique for finding infinite traces using the inequality-based analysis technique. Consider the simplest case where any infinite trace of a concurrent system is sought (as opposed to seeking an infinite trace with a specific property). In an infinite (augmented) trace, each transition either occurs a finite or an infinite number of times. There exists a finite prefix of the infinite trace containing all occurrences of transitions that occur only a finite number of times in the trace. I divide the trace into two intervals using such a prefix: the *finite interval* is a prefix of the trace containing all occurrences of transitions that occur only a finite number of times in the trace. The *perpetual interval* is the remainder of the trace, containing only transitions that occur infinitely often in the trace. Note that the transitions in a particular FSA that occur infinitely often in the trace must be part of a strongly connected component (SCC) in the FSA when viewed as a graph (i.e., to run forever, the FSA must traverse a cycle or set of interconnected cycles). An isolated state is not considered connected to itself unless there is an explicit self-loop, so these SCCs must contain at least one arc.

I construct an interval system for the finite interval and another for the perpetual interval and connect these systems together to form an inequality system representing necessary conditions for the existence of an infinite trace. The interval system for the finite interval is exactly the same as the one described in Section 5.1.1 except that this interval does not end with a particular event. Therefore, I assign connection variables to all states that are part of SCCs, allowing the finite part of an FSA's behavior to end at any point it could start repeating events forever. Also, I assign accepting variables to accepting states, allowing an FSA to have legal finite behaviors (the analysis technique does not handle finite behaviors by adding self-loops on stopped symbols, as the formalism does, but simply allows the flow through an FSA to exit via an accept variable before the perpetual interval). Finally, I do not generate requirement equations, since no events are required or forbidden.

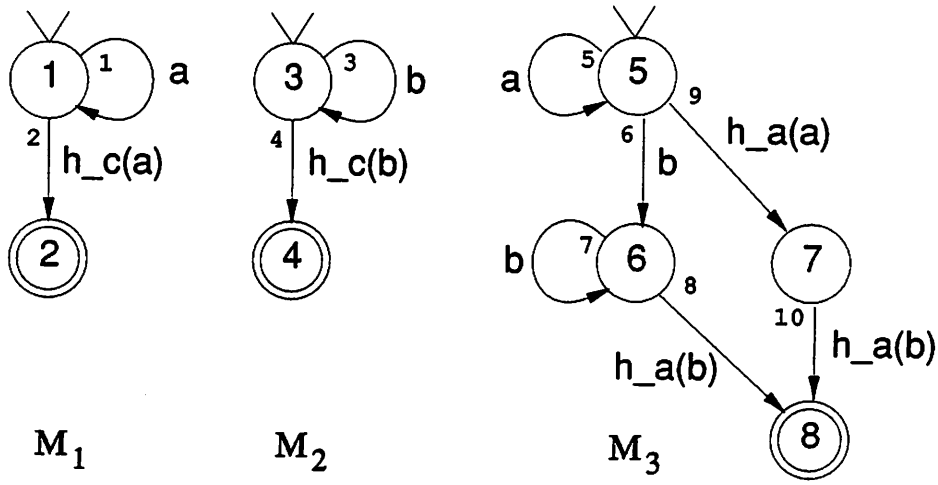


Figure 5.5. Example for Infinite Traces

The interval system for the perpetual interval is quite different. In this interval, any positive flow through an arc represents infinitely many occurrences of the event labeling that arc in the trace (the actual amount of flow does not contain any information about the relative frequencies of the events, nor the order in which they occur in the infinite trace—only that they occur infinitely often). I generate the perpetual interval system as follows. First, I conceptually remove parts of the FSAs not part of SCCs (in the example of Figure 5.1, which is repeated in Figure 5.5, I remove transitions 2, 4, 6, 8, 9, and 10, and ignore states 2, 4, 7, and 8). Second, I generate flow equations for the remaining states, but I do not add an implicit flow of one into the start states, nor do I count the connection variables from the finite interval as flow in. The connection between the finite and perpetual intervals is different than the connection between intervals in the technique of Section 5.1.1, as I will explain. Third, I generate synchronization equations for each channel equating the number of perpetual calls and accepts for that channel. Since hang symbols cannot occur perpetually, the perpetual interval system contains no hang inequalities. Also, unless the analyst specifies otherwise, there are no required or forbidden events, and thus no requirement inequalities.

I connect the finite and perpetual interval systems using additional inequalities. Unlike the way intervals were connected in Section 5.1.1, the flow through an FSA does not pass from the finite interval to the perpetual interval through a connection variable; the flow through the finite interval simply ends at some state in the FSA that is part of an SCC and then a cyclic flow (with no beginning or end) is forced to occur in the SCC as part of the perpetual interval. For each state j in an SCC, I add a *perpetual-force inequality* $\sum_{k \in out(j)}[x_{2,k}] \geq c_{1,j}$. I use the same subscripting for connection and transition variables as in Section 5.1.1: interval 1 is the finite interval, interval 2 is the perpetual interval. This inequality requires that if the FSA containing state j enters the perpetual interval at state j, then there must be flow through state j in the perpetual interval.

If the flow through an FSA in the finite interval exits via an accept variable rather than a connection variable, then no cyclic flows are forced to occur in that FSA. In fact, in this case I want to prevent such cyclic flows in the FSA. I could enforce this for an FSA N using the quadratic equation

$$\left(\sum_{k \in trans(N)}[x_{2,k}]\right)\left(1 - \sum_{j \in states(N)}[c_{1,j}]\right) = 0$$

where *states(N)* is the set of states of FSA N and *trans*(N) is the set of transitions in FSA N (note that $\sum_{j \in states(N)}[c_{1,j}] \leq 1$). In practice, since quadratic programming is much harder than linear programming, I achieve the desired relation using a linear inequality by using a large upper bound U for the transition variables (i.e., $x_{i,k} \leq U$). For each FSA N, I add a *perpetual-bound inequality*,

$$\left(\sum_{k \in trans(N)}[x_{2,k}]\right) \leq T \cdot U \left(\sum_{j \in states(N)}[c_{1,j}]\right)$$

where T is the number of transitions of N in an SCC (i.e., the number of transition variables on the left hand side of the inequality). If the flow through the FSA in the finite interval exits via an accept variable (i.e., $\sum_{j \in states(N)}[c_{1,j}] = 0$) then no flow is allowed in the FSA in the perpetual interval. If, however, the flow exits via a

connection variable (i.e., $\sum_{j \in states(N)}[c_{1,j}] = 1$), then this inequality allows any amount of flow (less than U) through arcs in the FSA in the perpetual interval.

I could strengthen the necessary conditions by generating a perpetual-bound inequality for each SCC, preventing flow through the SCC unless a connection variable into that SCC is nonzero. Since, in my experience, most task FSAs have only one maximal SCC, and for convenience in the implementation, I have chosen to automate generation of the weaker perpetual-bound inequalities described above.

The inequalities described are necessary conditions for the existence of a potentially infinite trace. The inequality system representing necessary conditions for the existence of a potentially infinite trace of the example of Figure 5.5 is shown in Figure 5.6. I may test for the possibility that M_2 becomes permanently blocked by adding the equation $x_{1,4} = 1$. The resulting inequality system has a solution corresponding to an infinite trace in which transition 4 is taken once and transitions 1 and 5 are taken perpetually ($x_{2,1} = x_{2,5} = 1$). This proves that the b communication need not eventually occur. Most systems would enforce some type of fairness in the selection of a communication partner that would prevent this behavior. I discuss the generation of inequalities to enforce fairness in Section 5.2.1.

The algorithm for the generation of an inequality system for a finite and a perpetual interval is given in Figure 5.7.

Theorem 5.2 *The inequality system generated by the algorithm in Figure 5.7 is a set of necessary conditions for the existence of a finite or an infinite trace.*

Proof. I consider the finite and the infinite cases in turn. Suppose there exists a finite trace t. Following the argument in the proof of Theorem 5.1, I can construct flows through the FSAs, exiting through accept variables, that satisfy the finite interval system. Setting the connection variables and all transition variables in the perpetual interval to zero satisfies both the perpetual interval system and the perpetual force and bound inequalities. This completes the proof in the case of a finite trace.

Flow (finite):			**(state)**
$1 + x_{1,1}$	$=$	$x_{1,1} + x_{1,2} + c_{1,1}$	(1)
$x_{1,2}$	$=$	f_2	(2)
$1 + x_{1,3}$	$=$	$x_{1,3} + x_{1,4} + c_{1,3}$	(3)
$x_{1,4}$	$=$	f_4	(4)
$1 + x_{1,5}$	$=$	$x_{1,5} + x_{1,6} + x_{1,9} + c_{1,5}$	(5)
$x_{1,6} + x_{1,7}$	$=$	$x_{1,7} + x_{1,8} + c_{1,6}$	(6)
$x_{1,9}$	$=$	$x_{1,10}$	(7)
$x_{1,8} + x_{1,10}$	$=$	f_8	(8)
Synchronization (finite):			**(channel)**
$x_{1,1}$	$=$	$x_{1,5}$	(a)
$x_{1,3}$	$=$	$x_{1,6} + x_{1,7}$	(b)
Hang (finite):			**(channel)**
$x_{1,2} + x_{1,9}$	$\leq$	1	(a)
$x_{1,4} + x_{1,8} + x_{1,10}$	$\leq$	1	(b)
Flow (perpetual):			**(state)**
$x_{2,1}$	$=$	$x_{2,1}$	(1)
$x_{2,3}$	$=$	$x_{2,3}$	(3)
$x_{2,5}$	$=$	$x_{2,5}$	(5)
$x_{2,7}$	$=$	$x_{2,7}$	(6)
Synchronization (perpetual):			**(channel)**
$x_{2,1}$	$=$	$x_{2,5}$	(a)
$x_{2,3}$	$=$	$x_{2,7}$	(b)
Perpetual-force:			**(state)**
$x_{2,1}$	$\geq$	$c_{1,1}$	(1)
$x_{2,3}$	$\geq$	$c_{1,3}$	(3)
$x_{2,5}$	$\geq$	$c_{1,5}$	(5)
$x_{2,7}$	$\geq$	$c_{1,6}$	(6)
Perpetual-bound:			**(FSA)**
$x_{2,1}$	$\leq$	$Uc_{1,1}$	(1)
$x_{2,3}$	$\leq$	$Uc_{1,3}$	(2)
$x_{2,5} + x_{2,7}$	$\leq$	$2U(c_{1,5} + c_{1,6})$	(3)

Figure 5.6. Inequality System for a Potentially Infinite Trace

Input: A set M of FSAs
Output: A set of inequalities

For each transition k in an FSA of M:
  Create transition variable $x_{1,k}$
  If k is in an SCC then
    Create transition variable $x_{2,k}$
For each state j in an SCC of an FSA of M:
  Create connection variable $c_{1,j}$
For each accepting state j of an FSA of M:
  Create accept variable f_j
For each interval $i = 1, 2$:
  For each state j of an FSA of M:
    Generate flow equation:

$$[1]_{i=1 \wedge start(j)} + \sum_{k \in in(j)} x_{i,k} = \sum_{k \in out(j)} x_{i,k} + [c_{i,j}] + [f_j]_{i=1}$$

For each channel c:
  For each interval $i = 1, 2$:
    Generate synchronization equation:

$$\sum_{k \in call(c)} x_{i,k} = \sum_{k \in accept(c)} x_{i,k}$$

  Generate hang inequality: $\sum_{k \in hang\_c(c)} x_{1,k} + \sum_{k \in hang\_a(c)} x_{1,k} \leq 1$
For each state j in an SCC of an FSA of M:
  Generate perpetual-force inequality: $\sum_{k \in out(j)} [x_{2,k}] \geq c_{1,j}$
For each FSA M_i in M:
  Let T equal the number of transitions of M_i in SCCs
  Generate perpetual-bound inequality:

$$\sum_{k \in trans(M_i)} [x_{2,k}] \leq T \cdot U \left(\sum_{j \in states(M_i)} [c_{1,j}] \right)$$

Figure 5.7. Algorithm for Finite and Perpetual Intervals

The necessary condition used for the existence of an infinite suffix of a trace is the existence of a set of cyclic flows through the FSAs in the perpetual interval such that these flows satisfy the synchronization equations (i.e., the number of perpetual calls to an entry equals the number of perpetual accepts of that entry). Multiple occurrences of perpetual symbols are allowed so that the synchronization equations can be satisfied if the loops are not exactly synchronized (e.g., if the loop calling an entry contains two calls while the loop accepting the entry contains only one, then the loop accepting the entry must be traversed twice to match every traversal of the loop calling the entry). I argue that the existence of such cyclic flows is a necessary condition for the existence of infinite traces. Consider an augmented infinite trace of the concurrent system, starting at a point after all occurrences of transitions occurring only a finite number of times. By the pigeonhole principle, one of the finite number of global states must be repeated infinitely often. Between any pair of occurrences of this repeated global state, the paths traversed by the FSAs are cycles along which the number of calls of each entry equals the number of accepts of that entry. Therefore, there exist cyclic flows through SCCs satisfying the synchronization equations.

Now suppose there exists an infinite (augmented) trace t. The trace t can be divided into a finite prefix t_* and an infinite suffix t_ω such that t_ω contains only transitions occurring infinitely often in t. Again, following the argument in the proof of Theorem 5.1, I can construct flows through the FSAs for t_* that satisfy the finite interval system. The flow through any FSA that terminates exits through an accept variable. The flow through an FSA that run forever exits through a connection variable, which must exist since the state of such an FSA at the end of the augmented prefix t_* must be in an SCC (otherwise the FSA could not repeat events forever in t_ω). I argued above that the perpetual interval system was a set of necessary conditions for the existence of an infinite suffix of a trace. Thus I can construct flows through the SCCs of the FSAs that satisfy the perpetual interval system. I must now show that the perpetual force and bound inequalities also hold for these two sets of flows.

In the infinite suffix of the augmented trace, a particular global state, say s, is visited infinitely often. Without invalidating the above arguments, I can split the trace such that s is the first global state of the augmented suffix t_ω. The cyclic flows found for t_ω in the above argument will satisfy the perpetual-force inequalities since these flows pass through the states of the FSAs in s (these are exactly the states where the flows for the finite interval exited through the connection variables). Similarly, the perpetual-bound inequalities will be satisfied since cyclic flows will occur only in FSAs that run forever in the trace. Thus there exists a solution to the inequality system corresponding to the infinite trace. □

For an asynchronous communication mechanism, I would generate the same communication inequalities for the perpetual interval as those described for the finite interval in Section 3.2. In the perpetual interval, these inequalities guarantee that if there are an infinite number of message receptions on a channel, then there must be an infinite number of message sends to that channel.

In Section 5.3, this technique is combined with the technique for answering starless queries described in Section 5.1 and experiments with this combined technique are presented in Section 7.2.2.

5.2.1 Fairness

Certain types of fairness in the selection of communication partners can be enforced using additional inequalities. In Ada, for example, multiple calls on the same entry by different tasks are queued—a very strong type of fairness. At this time, I do not know of a practical method to enforce such a strong fairness with inequalities, but enforcing weaker types of fairness in the analysis is often sufficient to verify liveness properties. The queuing of entry calls guarantees that a task cannot become permanently blocked waiting on an entry call if a call on that entry by another task is accepted infinitely often. I can enforce this weaker fairness property with a

quadratic inequality that forbids both of the following quantities from being nonzero simultaneously: the number of **hang_c** symbols for an entry call, and the number of perpetual **beg_rend** symbols for tasks calling that entry. If x and y represent these quantities, respectively, then the quadratic inequality $xy = 0$ forbids both x and y from being nonzero simultaneously. In practice, since quadratic programming is much harder than linear programming, I use a linear inequality instead. Since x and y must be integral, $x \leq 1$ (there can be at most one **hang_c** symbol per channel in the trace), and the ILP variables have a large upper bound U (so $y \leq U$), I can achieve this relation with the linear inequality $Ux + y \leq U$.

Ada does not enforce any fairness when selecting communication partners that are calling different entries in a **select** statement, but the technique described above can be used to enforce such fairness. In an FSA, a **select** statement is modeled with a state having multiple out transitions on communication event symbols. I could enforce fairness on the selection of a communication partner at such a state by adding an inequality for each partner forbidding the partner from becoming permanently blocked waiting for communication at this state if any of the transitions out of the state occur perpetually. Consider the example in Figure 5.5. M_3 could have been generated by the CEDL code in Figure 5.8. States 5 and 6 represent the **select** statement for the possible values of the variable **flag**. To enforce fairness at state 5, I add the inequalities $Ux_{1,2} + x_{2,5} \leq U$ for M_1 and $Ux_{1,4} + x_{2,5} \leq U$ for M_2. Note that since transitions 6 and 9 are not in an SCC, they cannot occur perpetually. Fairness need not be enforced at state 6 since only M_2 can communicate with M_3 at that state. If I add these inequalities to the system in Figure 5.6, along with the upper bounds on all the variables, then the resulting inequality system no longer has a solution where $x_{1,4} = 1$. This proves that M_2 cannot become permanently blocked, given that an FSA cannot starve waiting for a communication that is infinitely often possible. The inequality system does still have a solution with $x_{1,2} = 1$, corresponding to an infinite trace in which M_1 becomes permanently blocked.

```
task body three is
flag : boolean := true;
begin
  loop
    select
      when flag => accept A;
    or
      accept B;
      flag := false;
    end select;
  end loop;
end three;
```

Figure 5.8. CEDL Code for M_3

5.3 ω-star-less Queries

In this section, I combine the technique for star-less queries in Section 5.1 with the technique for representing infinite traces in Section 5.2 to produce a technique for answering star-less queries about potentially infinite traces.

To do this, I simply allow the last interval of a sequence of a star-less query to be perpetual. I connect the perpetual interval to the last finite interval of the sequence just as it was connected to the only finite interval in Section 5.2, except for the following: if the last finite interval ends with a specific event, then I assign connection variables for that interval only to states in SCCs where an FSA could be after the ending event (i.e., I assign a connection variable to a state only if it would be assigned a connection variable by both the procedure for sequences in Section 5.1.1 and the procedure for infinite traces in Section 5.2).

The technique that results from the combination of these two methods can generate necessary conditions for the existence of a potentially infinite trace matching a query specified as follows. An interval query is given by the following:

- an ending event (may be omitted)
- a set of required events (may be empty)

- a set of forbidden events (may be empty)
- a set of additional linear inequalities over the numbers of certain event symbols (may be empty)
- a flag indicating whether the interval is open (i.e., the ending event may occur more than once and the forbidden events are ignored)
- a flag indicating whether the interval is perpetual
- a flag indicating whether the interval is initial (i.e., FSAs must begin the interval in their initial states)
- a flag indicating whether the interval is final (i.e., FSAs must end the interval in an accepting state)

A sequence query is a list of interval queries, such that only the first may be initial and only the last may be either perpetual or final (but not both). An *ω-star-less query* is a set of sequence queries (conceptually disjoined).

Queries match traces or segments of traces. A segment t of an augmented trace matches a query q if:

- q is an ω-star-less query containing sequences $S_1, \ldots, S_n$ and t matches S_i for some $i = 1, \ldots, n$.
- q is a sequence query consisting of intervals $I_1, \ldots, I_n$ and t can be divided into subsequences $t_1, \ldots, t_n$ (t_n may be infinite) such that $t = t_1 t_2 \ldots t_n$ and t_i matches I_i for every $i = 1, \ldots, n$.
- q is an interval query and all of the following hold:
  - If q ends with event e then the last event of t is e.
  - If q requires events $e_1, \ldots, e_n$, then t contains at least one e_i event for $i = 1, \ldots, n$.

- If q forbids events $e_1, \ldots, e_n$, and if q is not open, then t contains no e_i events for $i = 1, \ldots, n$.
- If q specifies linear inequalities over the numbers of certain event symbols, then these relations hold in t.
- If q is not open, then its ending event (if specified) appears exactly once in t.
- If q is perpetual, then either t is empty or t is infinite. If q is not perpetual, then t is finite.
- If q is initial, then all FSAs are in their initial state at the beginning of t.
- If q is final, then all FSAs are in accepting states at the end of t.

The above formulation of ω-star-less queries is more powerful than the one given in [23], which defines an ω-star-less query as an ω-regular expression of the form:

$$\bigcup_{i=1}^{m} S_{i,0}^* e_{i,1} S_{i,1}^* e_{i,2} \ldots S_{i,n_i-1}^* e_{i,n_i} S_{i,n_i}^* T_i^\omega$$

where $S_{i,j} \subseteq \Sigma$, $e_{i,j} \in \Sigma$, $T_i \subseteq \Sigma$. A trace t satisfies such a query if it is in the language of strings generated by the expression. The formulation given here is more powerful because of the linear inequalities allowed on the total numbers of event symbols within intervals. These inequalities allow non-regular properties to be specified. For example, I can construct a query matching traces with an equal number of a's and b's. I call these more powerful ω-star-less queries *extended ω-star-less queries* to distinguish them from the ω-star-less queries defined with ω-regular expressions. I do not have a precise characterization of the expressibility of either ω-star-less queries or extended ω-star-less queries, but ω-star-less queries are less powerful than first order logic. This follows from the equivalence of first order logic and ω-star-free expressions in expressibility, the fact that ω-star-less queries are ω-star-free expressions of bounded dot-depth, and the strictness of the dot-depth hierarchy (which implies that there are ω-star-free expressions that are not expressible as ω-star-less queries) [73].

There is some question as to whether extended ω-star-less queries whose intervals do not specify arbitrary linear relations are equivalent in expressive power to ω-star-less queries. I believe that they are, but because I am concerned with the utility of the queries in practice, I have elected to spend time evaluating ω-star-less queries empirically rather than prove a theorem of questionable value. I know that extended ω-star-less queries can express certain common event patterns (e.g., the occurrence of certain symbols in no particular order) much more compactly than ω-star-less queries, and that, in these cases, they would produce smaller inequality systems which are likely to be easier to solve. This gives extended ω-star-less queries practical significance, even if less is known about their expressibility. In the sequel, I will refer exclusively to extended ω-star-less queries and drop the word "extended".

Section 7.2.2 presents a series of experiments with this technique in analyzing several variations of a moderately complex system. Although ω-star-less queries are clearly not powerful enough to express all regular or ω-regular languages, I have found them adequate to express many safety and liveness properties commonly verified. Section 5.4 discusses a technique that may be capable of processing any ω-regular query specified as a Büchi automaton. Although that technique would allow verification of any safety or liveness property expressible in linear-time temporal logic, it has not yet been implemented, and so its practical feasibility is unknown.

5.4 Regular and ω-Regular Queries

This section describes an untested technique for generating necessary conditions for the existence of a trace matching a query specified by an FSA or a Büchi automaton rather than an ω-star-less expression.

I first describe how to extend the technique of Section 5.1.1 to determine whether there exists a finitė trace of the system matching a regular query given by an FSA. I then describe how this can be combined with the technique of Section 5.2 to pose ω-regular queries given by a Büchi automaton. Büchi automata are known to be

more expressive than first order logic [62, 73]. Thus, at least in theory, any safety or liveness property that can be expressed in linear temporal logic can be verified using the technique. Since the technique is approximate, however, a practical test of the quality of the necessary conditions must be done before the technique can be considered significant.

I start with the finite case. Given a *query FSA* N with n states and alphabet Π, I want to generate necessary conditions for the existence of a finite trace t of the system such that $\rho_\Pi(t) \in L(N)$. As in the technique of Section 5.1.1, I restrict the query in the case that a communication between two tasks is represented by distinct event symbols in their respective FSAs. At most one event symbol representing each communication may appear in Π.

In the technique of Section 5.1.1, I essentially create a copy of the set of FSAs for each interval and then connect these copies with connection variables. Rather than picturing each copy of the FSAs as a distinct flowgraph, I can picture all the copies together as one large flowgraph by adding *connection arcs* between the states in different copies of the same FSA that share the same connection variable. For example, in the inequality system of Figure 5.3, I can interpret $c_{1,1}$ as the transition variable for an arc from the copy of state 1 in the first interval to the copy of state 1 in the second interval. This, in fact, is the internal representation used by the tools. A copy of all the FSAs, called a *phase*, is made for each interval and then connection arcs are added between these phases where appropriate. I will describe the technique in this section directly in terms of the flowgraph it creates and the additional inequalities generated to restrict the possible flows through this flowgraph (e.g., synchronization equations, requirement inequalities).

I process a query given by an FSA as follows. I first create a phase for each state in the query FSA. A node of the flowgraph is given by a pair (i, j) where i is the name of the state of the query FSA (i.e., the phase), and j is the name of the state in the system FSA (as usual, assume states of different system FSAs have distinct names).

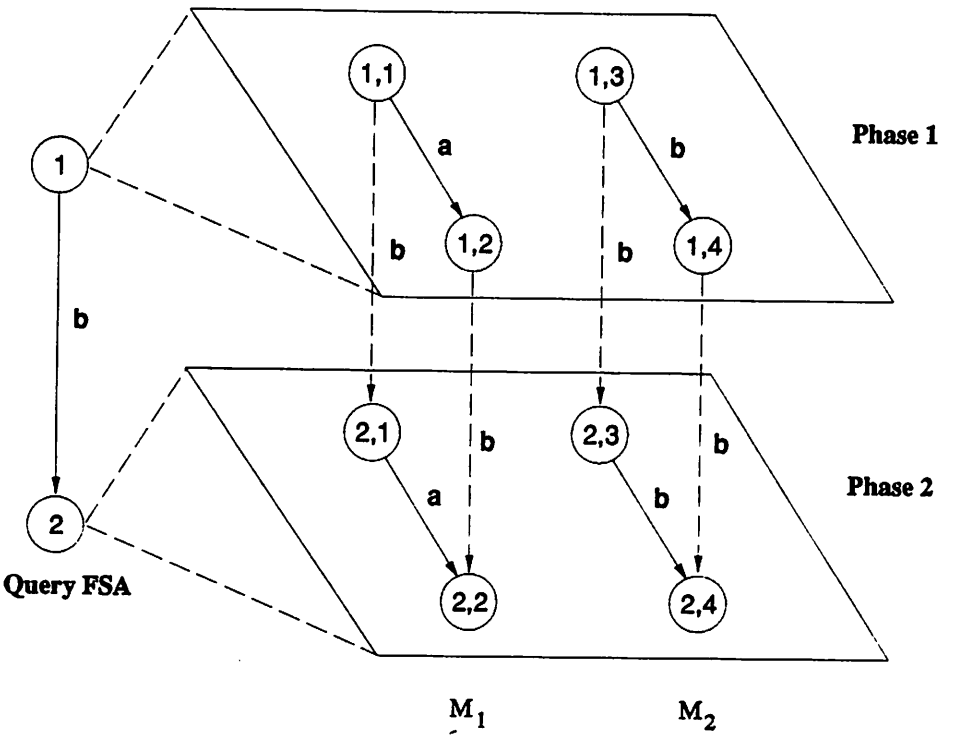


Figure 5.9. Visualizing the Flowgraph

I index the states of the query FSA with i, i' since these states act like the intervals of Section 5.1. I create a *transition arc* from node (i, j) to node (i, j') labeled with symbol e if there is a transition in a system FSA from state j to state j' on symbol e. I create a *connection arc* from node (i, j) to node (i', j) labeled with symbol e if there is a transition in the query FSA from state i to state i' on symbol e. The flowgraph created is simple to visualize in three dimensions. One can picture the system FSAs drawn on a plane and replicated, one above the next, for each phase. The copies of a given state are connected just as the corresponding states of the query FSA. An example is shown in Figure 5.9. Connection arcs are represented by dashed arrows.

I generate a flow equation for each node in the flowgraph. To nodes (i, j) where i is the starting state of the query FSA and j is a starting state of a system FSA, I add an implicit flow into the node of one. To nodes (i, j) where i is an accepting state of the query FSA and j is an accepting state of a system FSA, I add an accept variable, $f_{i,j}$, which is counted as flow out in the flow equation. I also generate transition

variables, accept variables, and flow equations for the query FSA in the standard way. Let $x_{(i,j),(i',j'),e}$ denote the transition variable for the arc from node (i,j) to node (i',j') labeled with event e and let $y_{i,i',e}$ denote the transition variable for the arc from state i to state i' labeled with event e in the query FSA. In addition to the flow equations, I generate the following additional inequalities:

- For each phase, I generate synchronization equations setting the number of times an entry is called equal to the number of times that entry is accepted *within that phase*, just as I did in Section 5.1.1 for each interval.

- For each communication, I generate a *hang* inequality involving the hang symbols from all the phases, just as I did in Section 5.1.1 for all the intervals.

- For each arc of the query FSA from state i to state i' labeled with symbol e, and for each system FSA M_j, I generate a *connection equation* setting the transition variable of this arc of the query FSA equal to the sum of the transition variables for the connection arcs labeled with e going from phase i to phase i' in this system FSA:

$$y_{i,i',e} = \sum_{j \in states(M_k)} x_{(i,j),(i',j),e}$$

  This equation forces a correspondence between the flow through the query FSA and the flow through the connection arcs of the flowgraph (e.g., if the query FSA goes from state 1 to state 2 on an a transition twice, then the flow in each system FSA must cross from phase 1 to phase 2 twice along a connection arc labeled a).

- For each state i of the query FSA and for each symbol e labeling a transition out of state i, I generate a *requirement equation* setting the number of occurrences of symbol e in phase i equal to the sum of the transition variables for arcs labeled with e leaving state i in the query FSA. This equation forces a correspondence between the flow through the query FSA and the flow through the transition arcs

Input: A set M of FSAs
A query FSA N
Output: A set of inequalities

For each state i of FSA N:
  For each state j of an FSA of M:
    Create node (i,j) in flowgraph F
  For each transition k of an FSA of M from state j to j':
    Create arc from (i,j) to (i,j') labeled with *label(k)* in F
For each transition k of FSA N from state i to i':
  Create transition variable $y_{i,i',label(k)}$
  For each state j of an FSA of M:
    Create arc from (i,j) to (i',j) labeled with *label(k)* in F
For each arc from node (i,j) to (i',j') on symbol e in F:
  Create transition variable $x_{(i,j),(i',j'),e}$
For each accepting state i of FSA N:
  Create accept variable f_i
  For each accepting state j of an FSA of M:
    Create accept variable $f_{i,j}$

Figure 5.10. Algorithm for FSA Query (Part 1)

of the flowgraph (e.g., if the query FSA leaves state 1 twice on an a transition, then there must be exactly two occurrences of event a in phase 1).

The algorithm for generating necessary conditions for the existence of a trace of the system matching a query specified by an FSA is given in Figures 5.10 and 5.11. Here, *arcs(G)* is the set of arcs in graph G.

Theorem 5.3 *The inequality system generated by the algorithm in Figures 5.10 and 5.11 is a set of necessary conditions for the existence of a finite trace t such that $\rho_\Pi(t) \in L(N)$.*

Proof. Suppose there exists a trace t matching the query: $\rho_\Pi(t) = e_1 e_2 \ldots e_n \in L(N)$. I describe a set of flows through the query FSA and the flowgraph satisfying the above inequalities. The sequence $e_1 e_2 \ldots e_n$ describes a path through the query FSA from

For each state i of FSA N:
  Generate flow equation:

$$[1]_{start(i)} + \sum_{(i',i,e)\in in(i)} y_{i',i,e} = \sum_{(i,i',e)\in out(i)} y_{i,i',e} + [f_i]$$

For each node (i,j) in F:
  Generate flow equation:

$$[1]_{start(i)\wedge start(j)} + \sum_{((i',j'),(i,j),e)\in in((i,j))} x_{(i',j'),(i,j),e} = \sum_{((i,j),(i',j'),e)\in out((i,j))} x_{(i,j),(i',j'),e} + [f_{i,j}]$$

For each state i in FSA N:
  For each channel c:
    Generate synchronization equation:

$$\sum_{((i,j),(i,j'),e)\in call(c)} x_{(i,j),(i,j'),e} = \sum_{((i,j),(i,j'),e)\in accept(c)} x_{(i,j),(i,j'),e}$$

For each channel c:
  Generate hang inequality:

$$\sum_{i\in states(N)} \left(\sum_{((i,j),(i,j'),e)\in hang\_c(c)} x_{(i,j),(i,j'),e} + \sum_{((i,j),(i,j'),e)\in hang\_a(c)} x_{(i,j),(i,j'),e} \right) \leq 1$$

For each transition from state i to i' of FSA N on event e:
  For each FSA M_k of M:
    Generate connection equation: $y_{i,i',e} = \sum_{j\in states(M_k)} x_{(i,j),(i',j),e}$

For each state i of FSA N:
  For each event e labeling a transition of state i:
    Generate requirement equation:

$$\sum_{(i,i',e)\in out(i)} y_{i,i',e} = \sum_{((i,j),(i,j'),e)\in arcs(F)} x_{(i,j),(i,j'),e}$$

Figure 5.11. Algorithm for FSA Query (Part 2)

its start state to one of its accepting states. A unit flow along this path will satisfy the flow equations for the query FSA. The (augmented) trace t describes a path through each system FSA from its starting state to one of its accepting states. For each system FSA, I describe a flow through the flowgraph corresponding to this path as follows. The flow starts at node (i, j) where i is the start state of the query FSA and j is the start state of the system FSA. It proceeds in the same phase through the transition arcs defined by the system FSA, following the path described by the trace, until just after event e_1. Let i be the state of the query FSA after event e_1 in the path through that FSA described above and let j be the state of the system FSA just after event e_1. The flow through the system FSA crosses the connection arc labeled e_1 from node (i, j) to node (i', j). The flow then proceeds in phase i' through the transition arcs defined by the system FSA, again following the path described by the trace, until event e_2. The flow continues in this way, alternating between following the path described by the trace through the transition arcs and changing phases through the connecting arcs after e_i events, until it reaches an accepting state in the phase corresponding to the state of the query FSA after event e_n. Note that along the flow for a given FSA that contains an e_i event, this event will occur twice in succession (once on a transition arc and once on a connection arc).

I now argue that these flows satisfy the necessary conditions generated by the algorithm in Figures 5.10 and 5.11. Clearly the flows satisfy the flow equations for the flowgraph and the query automaton. The trace is divided by the e_i events into intervals, just as in Section 5.1.1. The flows for each of these intervals are through some phase. The flows satisfy the synchronization equations for each phase since matching communication events in each interval will occur in the same phase. Given that the trace is legal, the flows must satisfy the hang inequalities. The connection equations are satisfied since the flow through a system FSA described above crosses a connection arc each time the corresponding transition in the query FSA is made. Finally, the requirement equations are satisfied because each time the flows leave a

phase over connection arcs labeled with an event e, it is immediately following an occurrence of event e in that phase. □

The inequality system generated by this technique is a set of necessary but not, in general, sufficient conditions for the existence of a trace of the concurrent system matching the query. The conditions are not sufficient for several reasons. First, just as the communication equations in Section 3.2 do not guarantee a globally consistent ordering of communication events for a trace, the synchronization equations here do not guarantee a globally consistent ordering of communication events in a particular interval of the trace.

Second, cycles in the system FSAs can create disconnected cyclic flows, as described in Section 3.2. I do not believe that cycles in the query FSA will cause this problem since there is no incentive to add these flows, especially when a minimal integral solution is sought. Cyclic flows in the system FSAs occur when communication events in the cycle are needed to match communication events on non-cyclic paths in other system FSAs. Since communication events in one phase cannot match those in another, however, there would be no point in adding an extra cyclic flow in the query FSA, which would force the occurrence of certain events in the system FSAs, since these extra events in the system FSAs could not match any communication events of the system FSAs in other phases.

A third and new reason why the conditions are not sufficient is that different intervals can share the same phase. This allows a communication event in one interval to match a communication event in another interval if the two intervals share the same phase. Also, if two intervals ending with different events share the same phase, these conditions allow flows through different system FSAs to select these ending events in different orders (e.g., two intervals, ending with a and b, share the same phase; the flow through M_1 exits the phase after an a, re-enters the phase, and then exits after

a b; the flow through M_2 exits the phase after a b, re-enters the phase, and then exits after an a).

These may not be the only reasons why the conditions are not sufficient. Experiments with the technique may reveal additional sources of spurious solutions.

I have shown how to generate necessary conditions for the existence of a finite trace t such that $t \in L(N)$ for an FSA N. I now extend this technique to generate necessary conditions for the existence of an infinite trace t such that $t \in L(N)$ for a Büchi automaton N. As described in Section 5.2, I can divide a potentially infinite trace into a finite part and an infinite part such that all the events occurring only finitely often appear in the finite part. I first describe how to generate necessary conditions for the finite part. I then describe how to generate necessary conditions for the infinite part. Finally, I describe how to combine these conditions into necessary conditions for the existence of an infinite trace $t \in L(N)$.

I may generate necessary conditions for the existence of a finite prefix of an infinite trace $t \in L(N)$ as follows. Treating N as an FSA, I use essentially the technique described above to generate the inequalities, except that:

- I assign a *perpetual variable*, $p_{i,j}$, to every node (i, j) in the flowgraph such that i and j are each part of an SCC of their respective automata. This variable will be one if the prefix ends with the system FSA containing state j in state j and the query automaton in state i, otherwise it will be zero. Perpetual variables are counted as flow out in the flow equations and act like the connection variables of Section 5.2, allowing the flow to exit the finite part at any point where both the system FSA and the query automaton could begin to loop forever.

- I also assign a perpetual variable, p_i, to each state i of N in an SCC. This variable will be one if the prefix ends with the query automaton in state i, otherwise it will be zero.

- I do not assign accept variables to states in N. The flow through the query automaton must exit via a perpetual variable since the query automaton must run forever (unlike the system FSAs, some of which may stop).

- I assign accept variables to all nodes (i, j) such that j is an accepting state of a system FSA (i.e., i does not have to be an accepting state of N as before).

Call the flowgraph generated for this (the finite) part of the trace F_*. The resulting inequality system represents necessary conditions for the existence of the finite prefix of a trace $t \in L(N)$.

I generate necessary conditions for the existence of an infinite suffix of a trace $t \in L(N)$ as follows. Again I treat N as an FSA and use the above technique, but with the following exceptions:

- I remove parts of the system FSAs not contained in SCCs.

- I remove parts of N not contained in SCCs to form the automaton N_ω.

- I do not add an implicit flow into any nodes and I do not add accept variables to any nodes in the flowgraph (I intend to find cyclic flows).

- I do not add an implicit flow into any states of N_ω and I do not add accept variables to any states of N_ω (again, I intend to find a cyclic flow).

- I add an *accept inequality* forcing at least one of the transitions out of an accepting state of N_ω to be taken, forcing a cyclic flow to pass through an accepting state.

Call the flowgraph generated for this (the infinite) part of the trace F_ω. The resulting inequality system represents necessary conditions for the existence of an infinite suffix of a trace $t \in L(N)$.

I combine the inequality systems for the finite and infinite parts of the trace by adding *perpetual-force* inequalities like the ones added in Section 5.2. Let $x'_{(i,j),(i',j'),e}$

be the transition variable for the arc from node (i,j) to node (i',j') labeled with event e in F_ω and let $y'_{i,i',e}$ be the transition variable for the arc from state i to state i' labeled with event e in N_ω. For each node (i,j) in F_ω, I generate the perpetual-force inequality

$$\sum_{((i,j),(i',j'),e)\in out((i,j))} x'_{(i,j),(i',j'),e} \geq p_{i,j}$$

which will force a cyclic flow to pass through node (i,j) in F_ω if the finite part of the trace ended with a system FSA in state j and N in state i. I also add a perpetual-force inequality

$$\sum_{(i,i',e)\in out(i)} y'_{i,i',e} \geq p_i$$

for each state i in N_ω. This forces a cyclic flow to pass through state i in N_ω if the finite part of the trace ended with N in state i.

The algorithm for generating necessary conditions for the existence of a trace of the system matching a query specified by a Büchi automaton is given in Figures 5.12–5.14. For an FSA M_i, $final(M_i)$ is the set of accepting states of M_i.

Theorem 5.4 *The inequality system generated by the algorithm in Figures 5.12–5.14 is a set of necessary conditions for the existence of an infinite trace $t \in L(N)$.*

Proof. Suppose there exists an augmented trace $t \in L(N)$. As in Section 5.2, I can divide t into a finite prefix t_* and an infinite suffix t_ω such that t_ω contains only transitions occurring infinitely often. I describe a set of flows through F_* and F_ω that corresponds to t and satisfies the above inequalities. For the finite part, t_*, I can describe a set of flows just as I did when proving the necessity of the conditions derived for finite traces. The only difference is where the flows end. The flow through N must exit through a perpetual variable rather than an accept variable. Since all the events in t_ω occur infinitely often, after accepting t_*, N must be in a state that is part of an SCC and therefore has a perpetual variable. The flow through a system FSA that reaches an accepting state and stops can exit through an accept variable,

Input: A set M of FSAs
A Büchi automaton N
Output: A set of inequalities

For each state i of FSA N:
  If i is in an SCC of N then
    Create perpetual variable p_i
  For each state j of an FSA of M:
    Create node (i,j) in F_*
    If i and j are in SCCs of their respective automata then
      Create perpetual variable $p_{i,j}$
    If j is an accepting state then
      Create accept variable $f_{i,j}$
  For each transition k of an FSA of M from state j to j':
    Create arc from (i,j) to (i,j') labeled with *label(k)* in F_*
For each transition k of FSA N from state i to i':
  Create transition variable $y_{i,i',label(k)}$
  For each state j of an FSA of M:
    Create arc from (i,j) to (i',j) labeled with *label(k)* in F_*
For each arc from node (i,j) to (i',j') on symbol e in F_*:
  Create transition variable $x_{(i,j),(i',j'),e}$

For each state i in an SCC of FSA N:
  For each state j in an SCC of an FSA of M:
    Create node (i,j) in F_ω
  For each transition k of an FSA of M from state j to j':
    Create arc from (i,j) to (i,j') labeled with *label(k)* in F_ω
For each transition k in an SCC of FSA N from state i to i':
  Create transition variable $y'_{i,i',label(k)}$
  For each state j in an SCC of an FSA of M:
    Create arc from (i,j) to (i',j) labeled with *label(k)* in F_ω
For each arc from node (i,j) to (i',j') on symbol e in F_ω:
  Create transition variable $x'_{(i,j),(i',j'),e}$

Figure 5.12. Algorithm for Büchi Automaton Query (Part 1)

For each state i of FSA N:
  Generate flow equation:

$$[1]_{start(i)} + \sum_{(i',i,e)\in in(i)} y_{i',i,e} = \sum_{(i,i',e)\in out(i)} y_{i,i',e} + [p_i]$$

For each node (i,j) of F_*:
  Generate flow equation:

$$[1]_{start(i)\wedge start(j)} + \sum_{((i',j'),(i,j),e)\in in((i,j))} x_{(i',j'),(i,j),e} = \sum_{((i,j),(i',j'),e)\in out((i,j))} x_{(i,j),(i',j'),e} + [f_{i,j}] + [p_{i,j}]$$

For each state i in FSA N:
  For each channel c:
    Generate synchronization equation:

$$\sum_{((i,j),(i,j'),e)\in call(c)} x_{(i,j),(i,j'),e} = \sum_{((i,j),(i,j'),e)\in accept(c)} x_{(i,j),(i,j'),e}$$

For each channel c:
  Generate hang inequality:

$$\sum_{i\in states(N)} \left(\sum_{((i,j),(i,j'),e)\in hang\_c(c)} x_{(i,j),(i,j'),e} + \sum_{((i,j),(i,j'),e)\in hang\_a(c)} x_{(i,j),(i,j'),e} \right) \leq 1$$

For each transition from state i to i' of FSA N on event e:
  For each FSA M_k of M:
    Generate connection equation: $y_{i,i',e} = \sum_{j\in states(M_k)} x_{(i,j),(i',j),e}$

For each state i of FSA N:
  For each event e labeling a transition out of state i:
    Generate requirement equation:

$$\sum_{(i,i',e)\in out(i)} y_{i,i',e} = \sum_{((i,j),(i,j'),e)\in arcs(F_*)} x_{(i,j),(i,j'),e}$$

Figure 5.13. Algorithm for Büchi Automaton Query (Part 2)

For each state i in an SCC of FSA N:
  Generate flow equation:

$$\sum_{(i',i,e)\in in(i)} y'_{i',i,e} = \sum_{(i,i',e)\in out(i)} y'_{i,i',e}$$

For each node (i,j) in F_ω:
  Generate flow equation:

$$\sum_{((i',j'),(i,j),e)\in in((i,j))} x'_{(i',j'),(i,j),e} = \sum_{((i,j),(i',j'),e)\in out((i,j))} x'_{(i,j),(i',j'),e}$$

For each state i in FSA N:
  For each channel c:
    Generate synchronization equation:

$$\sum_{((i,j),(i,j'),e)\in call(c)} x'_{(i,j),(i,j'),e} = \sum_{((i,j),(i,j'),e)\in accept(c)} x'_{(i,j),(i,j'),e}$$

For each transition from state i to i' of FSA N on event e:
  For each FSA M_k of M:
    Generate connection equation: $y'_{i,i',e} = \sum_{j\in states(M_k)} x'_{(i,j),(i',j),e}$

For each state i of FSA N:
  For each event e labeling a transition of state i:
    Generate requirement equation:

$$\sum_{(i,i',e)\in out(i)} y'_{i,i',e} = \sum_{((i,j),(i,j'),e)\in arcs(F_\omega)} x'_{(i,j),(i,j'),e}$$

Generate accept inequality $\sum_{i\in final(N)} \left(\sum_{(i,i',e)\in out(i)} y'_{i,i',e} \right) \geq 1$

For each node (i,j) in F_ω:
  Generate perpetual-force inequality:

$$\sum_{((i,j),(i',j'),e)\in out((i,j))} x'_{(i,j),(i',j'),e} \geq p_{i,j}$$

For each state i in an SCC of FSA N:
  Generate perpetual-force inequality: $\sum_{(i,i',e)\in out(i)} y'_{i,i',e} \geq p_i$

Figure 5.14. Algorithm for Büchi Automaton Query (Part 3)

since these variables are now on every node (i, j) where j is an accepting state of a system FSA. The flow through a system FSA that reaches an SCC and begins repeating events in that SCC forever can exit through a perpetual variable, since these variables are present on all nodes (i, j) where i and j are part of SCCs in their automata.

For the infinite part, I must find a set of cyclic flows through F_ω and N_ω. Define the *combined state* of a concurrent system and a query as the global state of the concurrent system plus the state of the query automaton. Since there are only a finite number of combined states, and yet the (augmented) trace t passes through infinitely many such states, at least one combined state must be visited infinitely often. Any segment t' of t lying between two occurrences of this repeated combined state describes a cycle in each system FSA and in N_ω. I can construct flows through F_ω for t' just as I constructed flows for t_* through F_* except that, since these flows are cyclic, I do not require any implicit flows or special variables to start or stop a flow.

Finally, I must show that the flows found above in N, N_ω, F_*, and F_ω satisfy the perpetual-force inequalities. If N begins repeating events in a state i, then at least one of the transitions out of state i must be taken infinitely often in t, and so some arc out of state i in N_ω must have nonzero flow. Thus the perpetual-force inequalities for states in N hold. If a system FSA begins repeating events when it is in state j and the query automaton is in state i, then at least one of the transitions out of state j must be taken infinitely often in t. This implies either that one of the transition arcs out of node (i, j) in F_ω must have nonzero flow, or that one of the connection arcs out of node (i, j) must have nonzero flow so that some transition arc out of a node for state j in F_ω can have nonzero flow. Thus the perpetual-force inequalities for the nodes of F_* hold. □

The techniques presented in this section have not yet been implemented. Therefore, their practical feasibility is unknown.

5.5 Critical Races

Critical races, described in Section 1.1, are often the source of errors in concurrent software. This section describes a way of extending the inequality-based analysis technique of Section 3.2 to prove the absence of a specific critical race from a concurrent program specification. Section 5.5.1 describes the basic technique, which can detect or prove the absence of a critical race between two tasks. Section 5.5.2 sketches a generalization of the technique to the detection of races involving more than two tasks. The results of several experiments with the basic technique are presented in Section 7.2.3.

5.5.1 Basic Technique

The technique involves proving that a syntactically possible race cannot actually occur. I define an *apparent race* as a state in a task FSA in which the task has the choice of communicating with several other tasks. Apparent races can be detected by searching the task FSAs for states having multiple out transitions on communication event symbols. A *real race* is an apparent race in which there actually exists an execution of the concurrent program where this apparent choice is real—at some point in the execution when this state is reached, more than one of the communications is possible. To determine if a state is a real race, I must reason about the possible executions of the concurrent program. I use the inequality-based technique as the basis of a method for generating a set of necessary conditions for an apparent race to be a real race. If these conditions are unsatisfiable, I have proved that the race is not real.

Races are of interest to concurrent programmers since they are the manifestation of the (conceptually) nondeterministic scheduler. Races are natural features of concurrent programs (e.g., two customers contend for a resource), but some races, usually called *critical races*, are the source of errors. These are real races that the programmer assumed were only apparent. If a concurrent program functions correctly only if a

specific task wins the race (i.e., is the first to communicate with the task having the choice), then the program can fail nondeterministically. The technique presented here can aid software developers in preventing this type of error by verifying that certain races are only apparent and therefore not critical.

I generate necessary conditions for an apparent race to be a real race as follows. Call the FSA containing the apparent race the *race FSA*. Let e_1 and e_2 be the *race symbols*—the event symbols denoting the communications involved in the race. If a race is real, then there exists an augmented prefix t of a trace such that te_1 and te_2 are also augmented prefixes. I use the technique of Section 5.1.1 to derive necessary conditions for the existence of a prefix of a trace ending with e_1. I do the same to derive necessary conditions for the existence of a prefix of a trace ending with e_2. These two sets of necessary conditions are interval systems, which I number 1 and 2. I connect these interval systems with necessary conditions for the actions of the FSAs to be the same in the two prefixes, except for the race symbols. Of these three steps, only the third requires elaboration.

An alternate technique for finding races involves generating three interval systems: one for t, and then two one-symbol interval systems for e_1 and e_2. Since the one-symbol interval systems would be very small (containing transition variables only for transitions on e_1 and e_2), it would seem that this technique would generate a much smaller inequality system. I have chosen not to use this technique for two reasons, both of which involve the implementation. First, unlike $t(e_1 \cup e_2)$, the expression $te_1 \cup te_2$ can be processed as a star-less query, saving code and simplifying the implementation. Second, optimizations to be described in Section 7.1.4 largely eliminate the difference in the sizes of the inequality systems produced by the two techniques.

For a critical race query, the analyst specifies the name of the race FSA and the two race symbols. The race FSA could contain several *race states*, states with an apparent race on the race symbols specified. To each such race state, j, I assign a

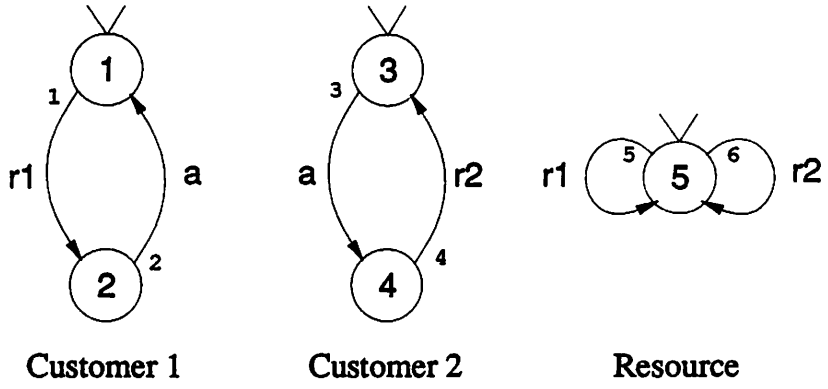


Figure 5.15. Alternation Example (Incorrect Version)

race variable, r_j, that will be one if I seek a real race at state j and zero otherwise. I generate a *race choice equation* summing these variables to one, forcing a race to be sought in exactly one race state. Let the transitions out of race states on race symbols be called *race transitions*. Let $x_{i,k}$ be the transition variable for transition k in interval system i (which finds a prefix of a trace ending with e_i). For each transition k not labeled with a race symbol, I generate the *race lock equation* $x_{1,k} = x_{2,k}$, forcing the number of times transition k is taken to be the same in the two prefixes. For each transition k in the race FSA labeled with race symbol e_1 (e_2) out of a race state with race variable r, I generate the race lock equation $x_{1,k} = x_{2,k} + r$ ($x_{1,k} + r = x_{2,k}$), forcing the transition on e_1 (e_2) in prefix 1 (2) to be taken one extra time if the race occurs at this state. For transitions on the race symbols outside the race FSA (i.e., not out of race states), I do not generate race lock equations. Since the race choice and race lock equations will force e_1 and e_2 to occur in their respective prefixes, I can omit the requirement equations from the corresponding interval systems.

I demonstrate the technique with a small example. A resource is to be used by two customers in strict alternation. I model this system with three FSAs, as shown in Figure 5.15. Event symbols $r1$ and $r2$ represent the use of the resource by customers 1 and 2 respectively; event symbol a represents a synchronous communication between the customers to enforce the alternation. Using the technique described above, I generate the system of inequalities shown in Figure 5.16 in an attempt to verify that

Flow (Interval 1):			**(state)**
$1 + x_{1,2}$	$=$	$x_{1,1}$	(1)
$x_{1,1}$	$=$	$x_{1,2} + c_{1,2}$	(2)
$1 + x_{1,4}$	$=$	$x_{1,3} + c_{1,3}$	(3)
$x_{1,3}$	$=$	$x_{1,4} + c_{1,4}$	(4)
$1 + x_{1,5} + x_{1,6}$	$=$	$x_{1,5} + x_{1,6} + c_{1,5}$	(5)
Synchronization (Interval 1):			**(channel)**
$x_{1,2}$	$=$	$x_{1,3}$	(a)
$x_{1,1}$	$=$	$x_{1,5}$	$(r1)$
$x_{1,4}$	$=$	$x_{1,6}$	$(r2)$
Flow (Interval 2):			**(state)**
$1 + x_{2,2}$	$=$	$x_{2,1} + c_{2,1}$	(1)
$x_{2,1}$	$=$	$x_{2,2} + c_{2,2}$	(2)
$1 + x_{2,4}$	$=$	$x_{2,3} + c_{2,3}$	(3)
$x_{2,3}$	$=$	$x_{2,4}$	(4)
$1 + x_{2,5} + x_{2,6}$	$=$	$x_{2,5} + x_{2,6} + c_{2,5}$	(5)
Synchronization (Interval 2):			**(channel)**
$x_{2,2}$	$=$	$x_{2,3}$	(a)
$x_{2,1}$	$=$	$x_{2,5}$	$(r1)$
$x_{2,4}$	$=$	$x_{2,6}$	$(r2)$
Race Choice:			**(state)**
r_5	$=$	1	(5)
Race Lock:			**(arc)**
$x_{1,2}$	$=$	$x_{2,2}$	(2)
$x_{1,3}$	$=$	$x_{2,3}$	(3)
$x_{1,5}$	$=$	$x_{2,5} + r_5$	(5)
$x_{1,6} + r_5$	$=$	$x_{2,6}$	(6)

Figure 5.16. Inequality System for Alternation Example (Incorrect Version)

there is no real race between $r1$ and $r2$ in the resource FSA. The first set of flow and synchronization equations are necessary conditions for the existence of a prefix of a trace ending with $r1$. The second set of flow and synchronization equations are necessary conditions for the existence of a prefix of a trace ending with $r2$. The race choice and race lock equations are necessary conditions for the actions of the FSAs in these two prefixes to be the same up to the race. Together, this forms a set of necessary conditions for the apparent race to be real. In this case, the inequality system has a solution (in which $x_{1,5} = 2, x_{2,5} = x_{2,6} = 1, x_{1,6} = 0$). This solution corresponds to the pair of prefixes $r1\ a\ r1$ and $r1\ a\ r2$, proving that the apparent race between $r1$ and $r2$ in the resource FSA is real. One synchronization between the customers is not enough to enforce strict alternation.

I modify the system by adding an additional synchronization between the customers, as shown in Figure 5.17. I again use the technique described above to generate necessary conditions for the existence of a real race between $r1$ and $r2$ in the resource FSA. This inequality system is shown in Figure 5.18 and has no integral solutions. This proves that the apparent race between $r1$ and $r2$ in this system *is not real*, and thus that the customers' protocol uniquely determines who may use the resource next.

The algorithm for generating necessary conditions for an apparent race to be real is shown in Figure 5.19. For a transition arc k, *from(k)* is the state that the transition arc comes from.

Theorem 5.5 *The inequality system generated by the algorithm in Figure 5.19 is a set of necessary conditions for an apparent race between the race symbols to be a real race.*

Proof. Suppose that the apparent race between e_1 and e_2 is real. Then there exist augmented prefixes te_1 and te_2. As I argued in the proof of Theorem 5.1, there exist solutions to the interval systems that correspond to these augmented prefixes. Let

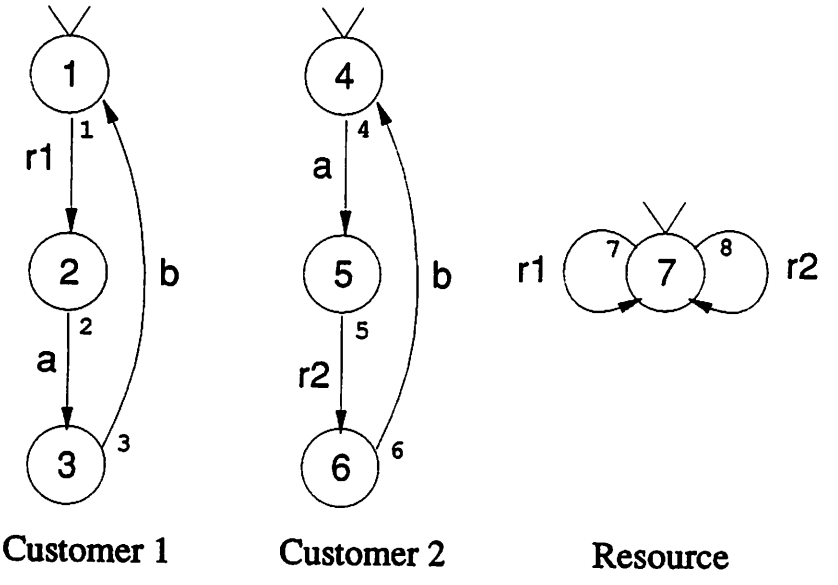


Figure 5.17. Alternation Example (Correct Version)

s be the state of the race FSA at the end of the augmented prefix t. I set the race variable for state s to one and all other race variables to zero. I show that the race choice and race lock equations are satisfied by the solutions corresponding to the prefixes together with this assignment of values to the race variables. Clearly the race choice equation is satisfied. The race lock equations for transitions not labeled with a race symbol must be satisfied since these transitions occur (if at all) in t, which is common to both prefixes. Next, I consider the race lock equations for race transitions not out of state s. These transitions also occur (if at all) only in t, and since the race variable in these equations has the value zero, the race lock equations will be satisfied as in the first case. Finally, I consider the race lock equations for the race transitions out of state s. These transitions are taken once more in one of the prefixes than in the other. The race lock equations for these transitions are satisfied, however, since the race variable in these equations has the value one and accounts for this difference. □

Flow (Interval 1):			**(state)**
$1 + x_{1,3}$	$=$	$x_{1,1}$	(1)
$x_{1,1}$	$=$	$x_{1,2} + c_{1,2}$	(2)
$x_{1,2}$	$=$	$x_{1,3}$	(3)
$1 + x_{1,6}$	$=$	$x_{1,4} + c_{1,4}$	(4)
$x_{1,4}$	$=$	$x_{1,5} + c_{1,5}$	(5)
$x_{1,5}$	$=$	$x_{1,6} + c_{1,6}$	(6)
$1 + x_{1,7} + x_{1,8}$	$=$	$x_{1,7} + x_{1,8} + c_{1,7}$	(7)
Synchronization (Interval 1):			**(channel)**
$x_{1,2}$	$=$	$x_{1,4}$	(a)
$x_{1,3}$	$=$	$x_{1,6}$	(b)
$x_{1,1}$	$=$	$x_{1,7}$	$(r1)$
$x_{1,5}$	$=$	$x_{1,8}$	$(r2)$
Flow (Interval 2):			**(state)**
$1 + x_{2,3}$	$=$	$x_{2,1} + c_{2,1}$	(1)
$x_{2,1}$	$=$	$x_{2,2} + c_{2,2}$	(2)
$x_{2,2}$	$=$	$x_{2,3} + c_{2,3}$	(3)
$1 + x_{2,6}$	$=$	$x_{2,4}$	(4)
$x_{2,4}$	$=$	$x_{2,5}$	(5)
$x_{2,5}$	$=$	$x_{2,6} + c_{2,6}$	(6)
$1 + x_{2,7} + x_{2,8}$	$=$	$x_{2,7} + x_{2,8} + c_{2,7}$	(7)
Synchronization (Interval 2):			**(channel)**
$x_{2,2}$	$=$	$x_{2,4}$	(a)
$x_{2,3}$	$=$	$x_{2,6}$	(b)
$x_{2,1}$	$=$	$x_{2,7}$	$(r1)$
$x_{2,5}$	$=$	$x_{2,8}$	$(r2)$
Race Choice:			**(state)**
r_7	$=$	1	(7)
Race Lock:			**(arc)**
$x_{1,2}$	$=$	$x_{2,2}$	(2)
$x_{1,3}$	$=$	$x_{2,3}$	(3)
$x_{1,4}$	$=$	$x_{2,4}$	(4)
$x_{1,6}$	$=$	$x_{2,6}$	(6)
$x_{1,7}$	$=$	$x_{2,7} + r_7$	(7)
$x_{1,8} + r_7$	$=$	$x_{2,8}$	(8)

Figure 5.18. Inequality System for Alternation Example (Correct Version)

Input: A set M of FSAs
A race FSA $N \in M$
A pair of distinct race symbols $e_1, e_2 \in \mathit{alphabet}(N)$
Output: A set of inequalities

For each interval $i = 1, 2$:
  For each transition k in an FSA of M:
    Create transition variable $x_{i,k}$
  For each state j in an FSA of M:
    If $\exists k \in \mathit{in}(j)(\mathit{label}(k) = e_i) \vee e_i \notin \mathit{alphabet}(\mathit{fsa}(j))$ then
      Create connection variable $c_{i,j}$
For each state j in FSA N:
  If $\exists k, k' \in \mathit{out}(j)(k \neq k' \wedge \mathit{label}(k) = e_1 \wedge \mathit{label}(k') = e_2)$:
    Create race variable r_j
For each interval $i = 1, 2$:
  For each state j of an FSA of M:
    Generate flow equation:
$$[1]_{\mathit{start}(j)} + \sum_{k \in \mathit{in}(j)} x_{i,k} = \sum_{k \in \mathit{out}(j)} x_{i,k} + [c_{i,j}]$$
  For each channel c:
    Generate synchronization equation: $\sum_{k \in \mathit{call}(c)} x_{i,k} = \sum_{k \in \mathit{accept}(c)} x_{i,k}$
Generate race choice equation: $\sum_{j \in \mathit{states}(N)} [r_j] = 1$
For each transition k in an FSA of M:
  If $\mathit{label}(k) \notin \{e_1, e_2\}$ then
    Generate race lock equation: $x_{1,k} = x_{2,k}$
  Else if $\mathit{fsa}(k) = N$ then
    If $\mathit{label}(k) = e_1$ then
      Generate race lock equation: $x_{1,k} = x_{2,k} + [r_{\mathit{from}(k)}]$
    Else
      Generate race lock equation: $[r_{\mathit{from}(k)}] + x_{1,k} = x_{2,k}$

Figure 5.19. Algorithm for Critical Race

5.5.2 *Multi-Way Races*

The technique described above allows the detection of a race between two specific communications. I now sketch an alternative technique for finding races between any number of communications. Rather than specifying two race symbols, I can specify a set of race symbols $\{e_1, \ldots, e_n\}$ representing communications. The technique can determine if any two of these communications can be involved in a race. The technique requires the use of an optimization to be described in Section 7.1.3.

As in the basic technique, I generate necessary conditions for the existence of two closely related prefixes, but rather than ending with a specific race symbol, each of these prefixes may end with any of the n race symbols $\{e_1, \ldots, e_n\}$. I then restrict the two prefixes to end with different race symbols. In the first race technique, the inequality system used to find each of the two prefixes was generated from an ω-star-less query having a single sequence consisting of a single interval ending with a specific race symbol. In the new technique, I generate the inequality system to find each prefix from an ω-star-less query consisting of n sequences each having one interval ending with a different e_i. Such a query matches a prefix of a trace ending with any one of the race symbols.

Let $s_{i,m}$ for $i = 1, 2,\ m = 1, \ldots, n$ be the sequence variable for the sequence ending with e_m in the inequality system for prefix i. These variables select the race symbols involved in the race. I generate an inequality for each $m = 1, \ldots, n$ stating that at most one of the prefixes can end with that race event: $s_{1,m} + s_{2,m} \leq 1$. These inequalities force the two prefixes to end with different race events. As before, I assign a race choice variable, r_j, to each race state j. These variables select the race state where the race occurs. I also assign a pair of *race arc variables*, $r_{i,k}$ for $i = 1, 2$, to each arc k in the race FSA labeled with a race event. Variable $r_{i,k}$ is one exactly when the last event in the prefix found by inequality system i is contributed by race transition k, and zero otherwise. These variables select the race transitions involved in the race. For each race arc variable, $r_{i,k}$, I generate a pair of *race bound inequalities*

to enforce the restrictions on the selection of race transitions given the selection of a race state and race symbols:

1. $r_{i,k} \leq r_{from(k)}$

2. $r_{i,k} \leq s_{i,m}$ where $label(k) = e_m$

An optimization in Section 7.1.3 allows multiple intervals from different sequences to share the same interval system. I use that optimization in this technique so that, in the inequality system generated for each prefix, each transition has one transition variable (not a different transition variable for each sequence in the query, as described in Section 5.1.2). When generating the race lock equations for the race transitions, I add the race arc variables instead of the race variables. Specifically, given that $x_{1,k}$ and $x_{2,k}$ are the transition variables for race transition k in the inequality systems for prefixes 1 and 2 respectively, I generate the race lock equation: $x_{1,k}+r_{2,k} = x_{2,k}+r_{1,k}$.

It is easy to show that exactly one race variable, r_j, and two race arc variables, $r_{1,k}$ and $r_{2,k'}$ for some $k, k' \in out(j)$, $k \neq k'$, will be one, forcing prefix 1 to end with $label(k)$ and prefix 2 to end with $label(k')$ with $label(k), label(k') \in \{e_1, \ldots, e_n\}$ and $label(k) \neq label(k')$.

5.6 Summary

The properties of interest to designers of concurrent software go beyond freedom from deadlock and other properties that can be expressed in terms of the numbers of certain events in a trace. This chapter has presented techniques that extend the basic constrained expression analysis technique of Section 3.2 to the verification of much more complex properties. Section 5.1 described how this inequality-based technique can be used to verify properties that involve the order of events, as do many important properties (e.g., mutual exclusion). Section 5.2 presented a technique for reasoning about infinite traces, allowing the verification of liveness properties. In Section 5.3, these two techniques were combined to form a technique for verifying

safety and liveness properties that can be expressed as ω-star-less queries, which are essentially regular or ω-regular expressions of a restricted form. Section 5.4 extended this technique to the verification of a larger class of properties, namely, those that can be expressed as arbitrary regular or ω-regular expressions, though this more powerful technique has not yet been tested. Finally, Section 5.5 presented a technique that can help developers detect critical races by determining if an apparent race can occur in any execution. The results of experiments with many of these techniques are presented in Chapter 7.

CHAPTER 6

DERIVING BOUNDS FOR REAL-TIME SYSTEMS

In this chapter, I address the verification of timing properties for real-time systems. Specifically, I give techniques to derive bounds on the time that can elapse between events in a concurrent real-time system running in a uniprocessor or maximally-parallel multiprocessor setting.

I model a concurrent real-time system using the formalism of Section 3.1 with the addition of a function $d : \Sigma \rightarrow R^+$ which assigns a nonnegative real duration to each event symbol. The definition of the time of a trace (or a segment of a trace) and the technique used to bound this time are dependent on the setting in which the concurrent program is run. I discuss the uniprocessor setting in Section 6.1 and the maximally-parallel multiprocessor setting in Section 6.2.

6.1 Uniprocessor Setting

This section describes a technique for deriving upper and lower bounds on the time that can elapse between two events in a concurrent real-time program run on a uniprocessor. It is based on an earlier, less automated technique described in [11]. Section 6.1.1 describes the technique for deriving bounds. Section 6.1.2 discusses cyclic flows, which present more of a problem in this analysis than in the analysis of concurrency properties in Chapter 5. Section 6.1.3 describes an optimization to the technique that can significantly reduce the size of the inequality system generated at the cost of possibly degrading the bounds obtained. An extension to account for periodic tasks in the analysis is described in Section 6.1.4. Finally, Section 6.1.5 elucidates my contributions by comparing the technique presented here to the earlier technique from [11]. The results of a series of experiments with the technique are presented in Section 7.2.4.

6.1.1 Basic Technique

In a uniprocessor setting, all events conceptually execute on the same processor so the duration of a trace (or segment of a trace) is simply the sum of the durations of the events involved. I assume that the concurrent system cannot deadlock, otherwise the duration of a segment of the trace between events could be unbounded, despite being represented by only finitely many event symbols. Deadlocks can be detected using the original concurrency analysis technique of Section 3.2 and must be removed prior to this analysis.

I can derive bounds on the duration of a segment of a trace beginning after a *start event* and ending with a *halt event* using the technique of Section 5.1.1. I generate inequalities for an ω-star-less query consisting of a sequence of two intervals: the first being an open interval ending with the start event, the second being an interval ending with the halt event. I seek a bound on the duration of the second interval (between the start and halt events), which I call the *timed interval.* I set the objective function of the inequality system equal to the duration of the timed interval, which is equal to the sum of the transition variables in the timed interval weighted by the duration of the event labeling the corresponding transition:

$$\sum_k d(label(k))x_{2,k}$$

where k ranges over all transitions in the FSAs.

Theorem 6.1 *Let A be the inequality system generated by the algorithm in Figure 5.4 when given the sequence of events e_1e_2 (making the first interval open). The maximum (minimum) value of the objective function $\sum_k d(label(k))x_{2,k}$ on the feasible region of A is an upper (lower) bound on the time that can elapse from just after the last occurrence of e_1 to just after the next occurrence of e_2 in any prefix of a trace of the concurrent system when run in a uniprocessor setting.*

Proof. Theorem 5.1 shows that A is a set of necessary conditions for the assignment of values to variables to correspond to a prefix of a trace of the form $t_1e_1t_2e_2$ for

$t_1 \in \Sigma^*$, $t_2 \in (\Sigma - \{e_1, e_2\})^*$. Since actions do not overlap on a uniprocessor, the time that elapses from just after the last occurrence of e_1 to just after the next occurrence of e_2 in such a prefix is equal to $\sum_k d(label(k))x_{2,k}$. Therefore, the maximum (minimum) value of this objective function on the feasible region of A is an upper (lower) bound on this time. The bound may not be sharp because the conditions represented by A are not sufficient. An optimal solution may not correspond to a prefix of a trace, but it will be at least as good as any solution that does correspond to a prefix of a trace, and thus yield a bound. □

Note that the timed interval cannot contain occurrences of the start event, nor can it contain any occurrences of the halt event except the one that ends the interval (this is the interpretation of the query described above as defined in Section 5.1.1). I accept this restriction for two reasons. First, it seems natural. Without this restriction, any system that repeats a pair of events forever would have no upper bound on the time that can elapse between those two events. The second reason is that this restriction makes the marking algorithm, to be described in Section 7.1.2, much more effective. The importance of this is discussed in the next section.

Also note that nothing is assumed about the scheduler on the uniprocessor, thus the bounds hold under any scheduling discipline. The bounds derived are for the worst case schedule.

6.1.2 Cyclic Flows

The marking algorithm, which will be described in Section 7.1.2, is vital for deriving quality upper bounds using this technique. Spurious cyclic flows, an occasional annoyance when doing concurrency analysis, are a nightmare when the integer programming package is instructed to maximize the objective function. Most systems contain sets of cycles whose communication event counts match. If a minimal solution is sought, no unnecessary flow will be induced through such sets of cycles. If a maximal

solution is sought, however, unbounded cyclic flows will be induced through these sets of cycles, yielding a useless upper bound of infinity.

There are two options for dealing with this problem. One is to remove transitions that cannot occur in the timed interval. I say that a cycle has been *broken* if an unbounded cyclic flow cannot arise through that cycle. If even one arc is removed from a cycle, then the cycle has been broken. In addition, cycles in other FSAs with communication events matching those in the broken cycle may now be broken. For example, the removal of the d cycle in M_2 of Figure 7.1 (page 153) breaks the d cycles in M_3, since there is no remaining d cycle in M_2 that would allow an unbounded cyclic flow through these cycles given the synchronization equation for d. The marking algorithm can automatically determine that certain transitions of the FSAs cannot occur in the timed interval. In addition, the analyst can supply additional information, in the form of required and forbidden events (or even additional start or halt events), to help the marking algorithm remove cycles of the FSAs that cannot occur in the timed interval.

The other option for dealing with cyclic flows is to bound the cycles directly using information not contained in the model. For example, the analyst may have a bound on the number of iterations of the loop that a cycle represents during the timed interval. Such a bound (unless zero) would not prevent spurious cyclic flows, but it could prevent unbounded cyclic flows and yield a finite upper bound.

6.1.3 Omitting the Initial Interval

The technique described generates necessary conditions for a prefix of a trace containing the start event followed by the halt event. I can generate a smaller system of inequalities to derive bounds on the time between events if I instead generate necessary conditions for a segment of a trace beginning after the start event and ending with the halt event. I can accomplish this using a query comprised of only one interval that ends with the halt event. I generate an inequality system for this

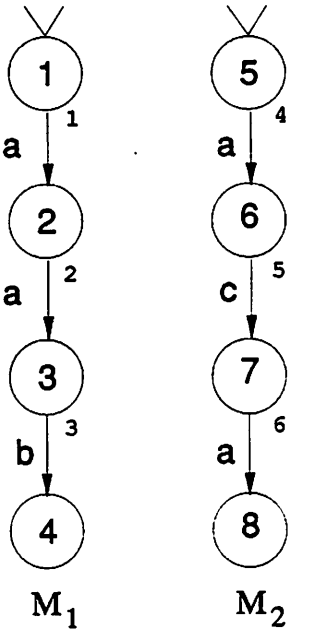


Figure 6.1. Example with Unreachable Segment

interval just as described in Section 5.1.1 except that I do not add an implicit flow into the start states of the FSAs. Instead, I assign a *start variable*, q_j, to every state j in which an FSA could be after the start event (i.e., every state that would be assigned a connection variable in an interval ending with the start event). This variable will be one if the FSA containing state j is in state j at the start of the segment of the trace, otherwise it will be zero. In the flow equations, start variables are counted as flow in. I generate a *start equation* for each FSA summing the start variables for that FSA to one, thus starting the flow through the FSA at some state in which the FSA could be after the start event. As an example, the inequality system generated for the concurrent system in Figure 6.1 is shown in Figure 6.2.

These inequalities are necessary conditions for the existence of a segment of a trace beginning after the start event and ending with the halt event. Since they are derived from only one interval, the inequality system will be roughly half the size of the one generated from the query for the prefix of the trace. This savings has its price, however, as these necessary conditions are weaker than those derived for the prefix and so the bounds derived may not be as good. Specifically, these conditions admit

Flow:			**(state)**
0	$=$	x_1	(1)
$q_2 + x_1$	$=$	x_2	(2)
$q_3 + x_2$	$=$	x_3	(3)
x_3	$=$	$c_{1,4}$	(4)
0	$=$	$x_4 + c_{1,5}$	(5)
$q_6 + x_4$	$=$	$x_5 + c_{1,6}$	(6)
x_5	$=$	$x_6 + c_{1,7}$	(7)
$q_8 + x_6$	$=$	$c_{1,8}$	(8)
Synchronization:			**(channel)**
$x_1 + x_2$	$=$	$x_4 + x_6$	(a)
Requirement:			**(symbol)**
$x_1 + x_2$	$=$	0	(a)
x_3	$=$	1	(b)
Start:			**(FSA)**
$q_2 + q_3$	$=$	1	(1)
$q_6 + q_8$	$=$	1	(2)

Figure 6.2. Inequality System to Find Segment from After a to b

more *unreachable segments*, segments whose initial global state is not reachable. An example of an unreachable segment for the system of Figure 6.1 starting after an a and ending with a b is given by transitions 3 and 5 in which M_1 starts in state 3 while M_2 starts in state 6. There is no trace of the concurrent system that can bring the FSAs to these states simultaneously. In this case, generating the necessary conditions for an initial interval would preclude this unreachable segment from being selected by the timed interval, since the initial interval could not contain two a events in M_1 but only one a event in M_2. In general, a solution to the inequality system generated for the initial interval need not correspond to a prefix of a trace. The presence of the initial interval does not necessarily prevent the timed interval from being an unreachable segment, though, in my experience, it usually does.

6.1.4 *Periodic Tasks*

I now describe an untested technique for dealing with a commonly occurring type of task in real-time systems. A *periodic task* is a task that is activated or created every p time units to perform some sequence of actions. I can represent a periodic task as a task consisting of an infinite loop whose body contains the actions performed by the periodic task. I enforce the period of the task by requiring that the body of the loop be executed at least $\lfloor \frac{r}{p} \rfloor$ times and at most $\lceil \frac{r}{p} \rceil$ times in an interval of time r. Let t be the expression representing the duration of the timed interval, used as the objective function in the technique described above. Let x be the transition variable of a transition in the loop that is taken exactly once in every iteration of the loop. Given a periodic task with period p, I want to require

$$x \geq \left\lfloor \frac{t}{p} \right\rfloor$$
$$x \leq \left\lceil \frac{t}{p} \right\rceil$$

I can enforce this by adding the following *periodicity inequalities*:

$$(x+1)p > t$$
$$(x-1)p < t$$

This method can also be applied to bound iterations of loops that execute with some minimum separation due to constraints from the environment. For example, a loop that processes disk interrupts might execute at most every 10 milliseconds simply due to the physical speed of the device. In such a case, only the inequality bounding the number of loop iterations from above would be added to the inequality system.

Up to this point, I have assumed that there is always an integral solution to an inequality system generated to obtain a uniprocessor bound. Without periodicity inequalities, this is the case provided that at least one trace contains the start

and halt events in order. The addition of periodicity inequalities that require a minimum number of executions of the periodic task can produce inconsistency if there is insufficient CPU time to accommodate all the required periodic computation. For example, a concurrent system with two periodic tasks, each with period 3 and duration 2, would have no traces containing an additional event of duration 3 and satisfying the periodicity restrictions specified by the inequalities described above. The inequality system generated for this system would have no integral solution. This can serve as a weak check on the feasibility of scheduling the tasks while satisfying the requirements of the periodic tasks.

6.1.5 Comparison with the Earlier Technique

The technique of Section 6.1.1 is based on an earlier technique developed by George Avrunin, Laura Dillon, and Jack Wileden [11]. I contrast these two techniques and clarify my contributions. The earlier technique used essentially the same method to derive the bounds: generate an inequality system representing necessary conditions for the existence of a segment of a trace between the start and halt events and use the optimal solution to this system as a bound. My principal contribution was completely automating the generation of the necessary conditions. The earlier technique required that the analyst manually edit the regular expressions representing the tasks of the concurrent system to allow only events that could occur between the start and halt events. Inequalities were then generated automatically from these altered expressions. The current technique utilizes the technology of Section 5.1.1 to generate necessary conditions for a segment of a trace automatically. This method alone, however, was found to be insufficient to derive useful upper bounds since it does nothing to prevent spurious cyclic flows. Therefore, I devised the marking algorithm of Section 7.1.2 to automatically remove cycles that can easily be determined to be unreachable in the segment between the start and halt events.

6.2 Multiprocessor Setting

This section describes a technique for calculating an upper bound on the time that can elapse between two events in a concurrent program running in a maximally-parallel multiprocessor setting. I begin in Section 6.2.1 by reviewing a well known technique for calculating the parallel execution time of a partially ordered set of events comprising a specific execution. In Section 6.2.2, I combine this technique with the uniprocessor bound technique of Section 6.1 to obtain a bound on the parallel execution time of a program over all of its executions. Section 6.2.3 describes how to generalize the technique to derive bounds on the time between events (rather than on whole executions). Finally, in Section 6.2.4, I describe some methods for tightening the bound derived by the technique. The results of several experiments with the technique are presented in Section 7.2.5.

6.2.1 Calculating Parallel Execution Time

The chief difficulty in extending the technique of Section 6.1 to the multiprocessor setting is calculating the parallel execution time of a set of events. On a uniprocessor, the execution time is simply the sum of the durations of the events that occur. On a multiprocessor, however, some events may happen concurrently and the execution time depends not only on which events occur, but on the synchronization structure created by those events.

By restricting attention to the case where each task has its own processor, I can use a well-known technique from scheduling theory to calculate the parallel execution time. Given a set of actions to be completed, each with a duration, and a partial order ($\leq$) on these actions representing a *wait* relation (i.e., if $A \leq B$ then action A must complete before action B can start), the following is well known: The minimum time to complete all the actions, assuming an action can proceed once ready, is equal to the length of the longest path, called the *critical path*, through the graph of the wait relation. Here the length of a path is the sum of the durations of the actions

along the path. If the durations of the actions are known, the critical path can be found in time linear in the size of the wait relation by successively marking actions with the earliest time they could commence, starting with actions that do not wait for other actions and proceeding up the partial order.

Given a specific execution of a concurrent program (which can be viewed as a set of "straight-line" FSAs), I can calculate the execution time in a maximally-parallel setting using this technique. Events in the concurrent program model correspond to actions in the technique. Event A immediately precedes event B in the partial order if either A and B are in the same FSA and A immediately precedes B in that FSA, or A is a communication event and B is the event following the matching communication event in the other FSA. The graph of this partial order for a trivial example is shown in Figure 6.3. M_1 executes internal action a, synchronizes with M_2 on channel b, and then M_2 executes internal action c. I use arcs to represent actions and nodes to connect the actions. This graph is the dual of the usual graph of a partial order, in which actions are represented by nodes, but it allows the FSAs to be part of the wait graph. Arcs in the wait graph that come from transitions in the FSAs are called *transition arcs*, while arcs that come from the matching of communication events are called *cross arcs* and are shown as dashed arrows. The critical path, assuming all events take unit time, is shown by the bold arrows in the figure.

6.2.2 Obtaining a Bound Over All Executions

Unfortunately, tasks in real concurrent systems are more complicated than those in the example of Figure 6.3. Allowing branches in the tasks makes static determination of matching communication events impossible since the matching may depend on what branches are taken. Loops in tasks introduce cycles in their FSAs and the wait graph, making the above algorithm, which applies only to partial orders, inapplicable. Scheduling theorists, to my knowledge, have not addressed the problem of finding the minimum time to complete a set of actions when the structure of the

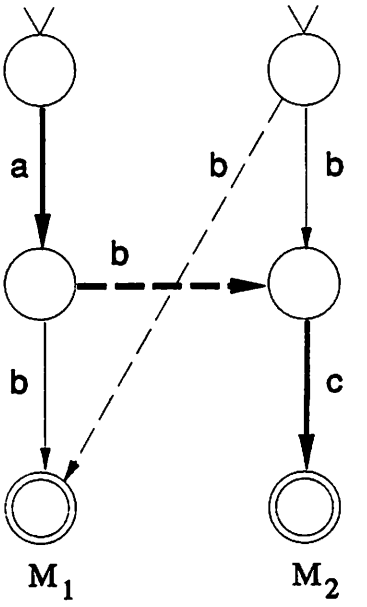


Figure 6.3. Wait Graph Showing Critical Path

wait relation varies. Rather, their work has focussed on the more common problem of finding expected completion times when the wait relation is fixed, but the durations of the actions are given by random variables having distributions of a specific form (e.g., [32]). My technique assumes fixed upper bounds on action durations and derives an upper bound on the minimum time to complete the actions given a variable wait relation that reflects the different possible executions of the concurrent program.

I first present an overview of the technique and then describe it in detail. Just as in the uniprocessor technique, I derive a set of linear inequalities from the FSAs and the semantics of the model. The optimal integral solution to this system is a bound on the execution time. The inequality system I derive for the multiprocessor technique has three parts. The *execution part* is similar to the inequality system generated for the uniprocessor technique described in Section 6.1. These equations "find" an execution of the concurrent program (i.e., choose what branches are taken, how many times each loop is traversed, etc.). The *critical path part* of the inequality system is a set of equations derived from the *potential wait graph*, described below. These equations "find" a path, which I call the *wait path*, through the potential wait

graph. For any critical path of an execution, there will be a corresponding wait path through the potential wait graph having the same length. The *bounding part* of the inequality system is a set of inequalities that bounds the variables from the critical path part with variables from the execution part, forcing the wait path found by the critical path part to pass through only events that actually occurred in the execution found by the execution part. Maximizing the length of the wait path will give an upper bound on the length of the critical path over all possible executions. This is an upper bound on the parallel execution time.

The potential wait graph, W, is formed from the FSAs by assuming each communication event could be paired with any syntactically possible match and adding a pair of cross arcs for each possible match (as was done for the single possible match in Figure 6.3). A cross arc is labeled with the communication symbol of the transition arcs it matches (if these arcs have different symbols denoting the communication, then it is labeled with either symbol). This graph represents all possible wait relations and also some that are not possible. If this graph contains cycles (as it will if the tasks contain loops), a wait path through the graph may cross certain arcs multiple times; each traversal of an arc represents a different occurrence of the corresponding event. The number of traversals will usually be bounded by the execution part.

I now describe in detail the generation of inequalities for obtaining a bound on the execution time of a concurrent program run in a maximally parallel multiprocessor setting. As noted above, the inequality system has three parts. The equations generated for the execution part are, in the terminology of Section 5.3, the same as those generated for an interval that is both initial and final (i.e., flow begins at start states and ends at accepting states) and forbids hang symbols (thus no hang inequalities are generated). This inequality system is a set of necessary conditions for the existence of a finite trace of the concurrent system in which none of the tasks become permanently blocked.

The equations generated for the critical path part find a flow through the potential wait graph, representing the wait path, from a starting state of some FSA to an accepting state of some FSA. (Since the nodes of the potential wait graph are the states of the FSAs, I call them states to simplify the description). To each arc k in the potential wait graph, I assign a *wait path variable*, y_k, that represents the number of times that arc is traversed as part of the wait path. To the start state j of each FSA, I assign a *begin variable*, b_j, that will be one if the wait path begins at that state, and zero otherwise. Similarly, to each accepting state j of an FSA, I assign a *halt variable*, h_j, that will be one if the wait path ends at that state, and zero otherwise. I then generate flow equations for each state in the potential wait graph, counting begin variables as flow in and halt variables as flow out. Finally, I generate one *begin equation* summing the begin variables to one, thus forcing the wait path to begin at exactly one state.

The inequalities generated for the bounding part bound the wait path variables from the critical path part with the transition variables from the execution part. I generate an *arc bound inequality* for each transition arc k that bounds the wait path variable for k with the transition variable for the corresponding transition: $y_k \leq x_k$. This insures that the wait path will not pass through a transition arc any more times than that transition occurred in the execution. I also bound the wait path variables for cross arcs, each of which represents the matching of communication events on two transitions in different FSAs. I generate a pair of arc bound inequalities for cross arc k, representing the matching of communication events on transitions k' and k'', that bounds the wait path variable for k with the transition variables for k' and k'': $y_k \leq x_{k'}$, $y_k \leq x_{k''}$. This insures that the wait path will not pass through a cross arc (which represents the occurrence of a communication event) any more times than the transitions that the arc matches occur in the execution. Finally, for each channel, I generate a *communication bound inequality* that bounds the number of occurrences of communication events for that channel in the wait path with the number of

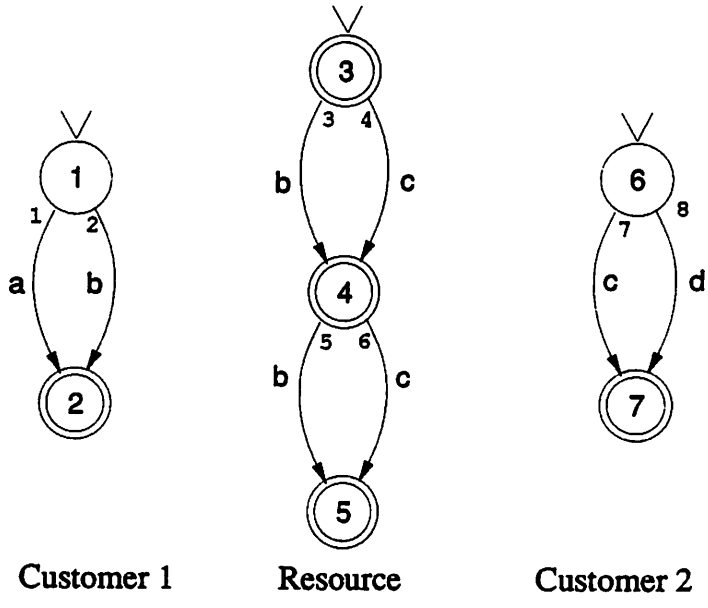


Figure 6.4. Resource Contention Example

occurrences of the communication in the execution. If a communication occurs once in the timed interval (a common case), this inequality insures that the wait path passes through at most one arc representing the occurrence of that communication.

For the objective function, I use the sum of the durations of the events occurring on the wait path, which is given by $\sum_{k \in arcs(W)} d(label(k))y_k$. To prevent anomalies in the selection of the wait path, I require that matching communication events have the same duration (i.e., for every channel c: $e_1, e_2 \in symbols(c) \Rightarrow d(e_1) = d(e_2)$, where $symbols(c)$ is the set of event symbols used to represent communication on channel c).

I demonstrate the technique with the small example of Figure 6.4. Two customers each choose whether to use a common resource that can be used at most twice. The use of the resource is represented by a synchronous communication between the customer and the resource. I seek a bound on the time for all tasks to complete.

The potential wait graph for this system is shown in Figure 6.5. The inequality system generated for this example is shown in Figure 6.6.

If all events take unit time, then the upper bound derived is 2, produced by a solution in which both customers try to use the resource and must contend. If the

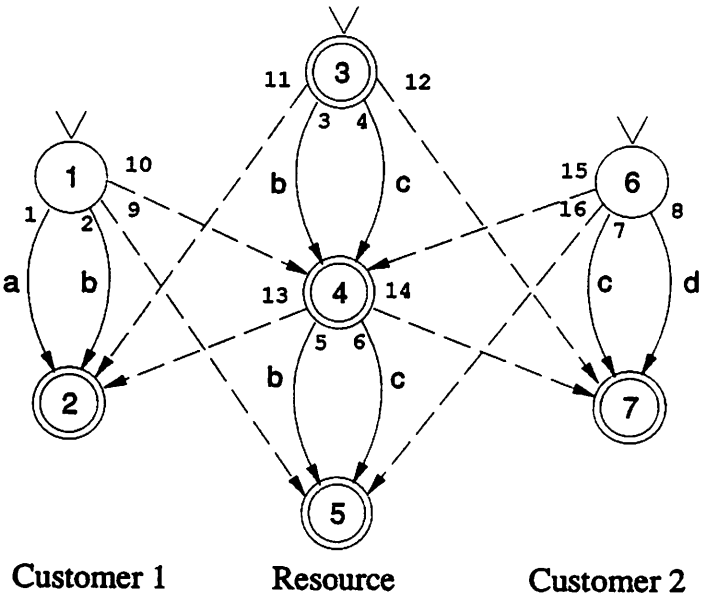


Figure 6.5. Potential Wait Graph for Resource Contention Example

b event takes 10 units instead, then the bound derived is 11, produced by the same solution. Note that the critical path part alone would yield a bound of 20 by selecting a path that does not correspond to a feasible execution (i.e., one passing through two *b* events in the resource task). If event *a* takes 10 units and the other events take only one, then the bound derived is 10, produced by a solution in which customer 1 does not use the resource.

The algorithm for generating a set of inequalities to derive multiprocessor bounds is shown in Figures 6.7 and 6.8. For a transition arc k, *from(k)* and *to(k)* are the nodes (states) that the transition arc comes from and goes to, respectively. For a graph G, *nodes(G)* is the set of nodes of graph G.

Theorem 6.2 *The maximum value of the objective function* $\sum_{k \in arcs(W)} d(label(k))y_k$ *on the feasible region of the inequality system generated by the algorithm in Figures 6.7 and 6.8 is an upper bound on the execution time of the concurrent system in a maximally-parallel multiprocessor setting.*

Proof. Let E be an execution of the concurrent system. I construct a solution to the inequality system having an objective value equal to the parallel execution

Flow (execution): **(state)**

$$
\begin{aligned}
1 &= x_1 + x_2 && (1)\\
x_1 + x_2 &= f_2 && (2)\\
1 &= x_3 + x_4 + f_3 && (3)\\
x_3 + x_4 &= x_5 + x_6 + f_4 && (4)\\
x_5 + x_6 &= f_5 && (5)\\
1 &= x_7 + x_8 && (6)\\
x_7 + x_8 &= f_7 && (7)
\end{aligned}
$$

Synchronization: **(channel)**

$$
\begin{aligned}
x_2 &= x_3 + x_5 && (b)\\
x_7 &= x_4 + x_6 && (c)
\end{aligned}
$$

Flow (critical path): **(state)**

$$
\begin{aligned}
b_1 &= y_1 + y_2 + y_9 + y_{10} && (1)\\
y_1 + y_2 + y_{11} + y_{13} &= h_2 && (2)\\
b_3 &= y_3 + y_4 + y_{11} + y_{12} + h_3 && (3)\\
y_3 + y_4 + y_{10} + y_{15} &= y_5 + y_6 + y_{13} + y_{14} + h_4 && (4)\\
y_5 + y_6 + y_9 + y_{16} &= h_5 && (5)\\
b_6 &= y_7 + y_8 + y_{15} + y_{16} && (6)\\
y_7 + y_8 + y_{12} + y_{14} &= h_7 && (7)
\end{aligned}
$$

Begin:

$$b_1 + b_3 + b_6 = 1$$

Arc Bound: **(arc)**

$$
\begin{aligned}
y_1 &\le x_1 && (1) & \qquad y_{11} &\le x_2 && (11)\\
y_2 &\le x_2 && (2) & \qquad y_{11} &\le x_3 && (11)\\
y_3 &\le x_3 && (3) & \qquad y_{12} &\le x_4 && (12)\\
y_4 &\le x_4 && (4) & \qquad y_{12} &\le x_7 && (12)\\
y_5 &\le x_5 && (5) & \qquad y_{13} &\le x_2 && (13)\\
y_6 &\le x_6 && (6) & \qquad y_{13} &\le x_5 && (13)\\
y_7 &\le x_7 && (7) & \qquad y_{14} &\le x_6 && (14)\\
y_8 &\le x_8 && (8) & \qquad y_{14} &\le x_7 && (14)\\
y_9 &\le x_2 && (9) & \qquad y_{15} &\le x_4 && (15)\\
y_9 &\le x_5 && (9) & \qquad y_{15} &\le x_7 && (15)\\
y_{10} &\le x_2 && (10) & \qquad y_{16} &\le x_6 && (16)\\
y_{10} &\le x_3 && (10) & \qquad y_{16} &\le x_7 && (16)
\end{aligned}
$$

Communication bound: **(channel)**

$$
\begin{aligned}
x_2 &\ge y_2 + y_3 + y_5 + y_9 + y_{10} + y_{11} + y_{13} && (b)\\
x_7 &\ge y_4 + y_6 + y_7 + y_{12} + y_{14} + y_{15} + y_{16} && (c)
\end{aligned}
$$

Figure 6.6. Inequality System for Resource Contention Example

Input: A set M of FSAs
Output: A set of inequalities

Remove all transitions on hang symbols from FSAs in M
For each transition k in an FSA of M:
  Create transition variable x_k
For each accepting state j of an FSA of M:
  Create accept variable f_j
For each state j of an FSA of M:
  Generate flow equation: $[1]_{start(j)} + \sum_{k \in in(j)} x_k = \sum_{k \in out(j)} x_k + [f_j]$
For each channel c:
  Generate synchronization equation: $\sum_{k \in call(c)} x_k = \sum_{k \in accept(c)} x_k$

Figure 6.7. Algorithm for Multiprocessor Bound (Part 1)

time of E in a maximally-parallel multiprocessor setting. It follows that a solution to the inequality system maximizing this objective function yields an upper bound on the duration of any execution of the concurrent system in a maximally-parallel multiprocessor setting.

The proof of Theorem 5.1 shows that there exists a solution to the execution part of the inequality system corresponding to E. As noted above, a particular execution of the concurrent system can be viewed as a set of actions and a wait relation, which is a partial order on these actions. Every execution has at least one critical path $e_1 \leq e_2 \leq \ldots \leq e_n$ through the graph of its particular wait relation such that for each $i = 1, \ldots, n-1$ either:

- e_i and e_{i+1} are actions of the same task and, in the execution, e_{i+1} immediately follows e_i in that task.

- e_i and e_{i+1} are actions of different tasks, and, in the execution, e_i matches a communication action e_i' in the task containing e_{i+1} and e_i' immediately precedes e_{i+1} in that task.

For each state j in an FSA of M:
  Create node j in the potential wait graph W
For each transition k in an FSA of M:
  Create transition arc k in W from node $from(k)$ to node $to(k)$
  labeled with $label(k)$
For each channel c:
  For each transition $k \in call(c)$:
    For each transition $k' \in accept(c)$:
      Create the following cross arcs in W matching transitions k and k':
        From node $from(k)$ to node $to(k')$ labeled with $label(k)$
        From node $from(k')$ to node $to(k)$ labeled with $label(k')$
For each arc k in W:
  Create wait path variable y_k
For each start state j of an FSA of M:
  Create begin variable b_j
For each accepting state j of an FSA of M:
  Create halt variable h_j
For each node j in W:
  Generate flow equation: $[b_j] + \sum_{k \in in(j)} y_k = \sum_{k \in out(j)} y_k + [h_j]$
Generate the begin equation: $\sum_{j \in nodes(W)} [b_j] = 1$

For each transition arc k in W:
  Generate arc bound inequality: $y_k \leq x_k$
For each cross arc k in W matching transitions k' and k'':
  Generate arc bound inequalities: $y_k \leq x_{k'}$ and $y_k \leq x_{k''}$
For each channel c:
  Generate communication bound inequality:
  $$\sum_{k \in arcs(W)} [y_k]_{label(k) \in symbols(c)} \leq \sum_{k \in call(c)} x_k$$

Figure 6.8. Algorithm for Multiprocessor Bound (Part 2)

I can construct a wait path through the potential wait graph labeled with this same sequence of actions (events). Each action in the wait relation corresponds to a transition of an FSA in the execution. Let t_i be the FSA transition corresponding to e_i. Since durations are nonnegative, I can always start a critical path in a start state of an FSA and end it in an accepting state of an FSA by adding extra events occurring in the execution to the path. The wait path starts in the start state with transition t_1. It then proceeds through the potential wait graph, following the transition arc for t_i if e_i and e_{i+1} are in the same FSA, and following the cross arc from the state with transition t_i to the state with transition t_{i+1} otherwise (if e_i and e_{i+1} are actions of different FSAs, then e_i must be a communication event, by the definition of the partial order, and so this cross arc must exist). The wait path ends in the accepting state with incoming transition t_n.

The wait path constructed is clearly a solution to the critical path part of the inequality system. This path satisfies the bounding part of the inequality system as well since the wait path is constructed from a sequence of distinct event occurrences in the execution. Each time the wait path traverses a transition arc, the corresponding transition in the FSA was taken once in the execution. Similarly, each time the wait path traverses a cross arc, the matching communication transitions were taken once in the execution. Finally, each time the wait path traverses a communication event, that event occurred once in the execution.

The value of the objective function at the solution described above is equal to the length of the wait path, which is equal to the length of the critical path, which is equal to the parallel execution time of the program in a maximally-parallel multiprocessor setting. □

The bound obtained by this technique need not be the least upper bound for several reasons. As described in Section 3.2, the conditions used in the execution part are only necessary and not sufficient, so the solution to this part of the inequality

system might not correspond to any possible execution. Also, cycles in the potential wait graph may allow cyclic flows in the solution that are not part of the wait path found. An example of this will be given in the remote server system of Section 7.2.5. In addition, the potential wait graph represents all syntactically possible matchings of communication events with cross arcs. Some of these matchings might not be possible and could produce spurious wait paths. An example of this will be given in the reactor monitor system of Section 7.2.5. Finally, the necessary conditions described do not strictly enforce maximal parallelism (i.e., no unnecessary waiting). When a task has a choice among several communication partners, it should communicate with the first partner to become ready and select one (arbitrarily) in case of a tie. The execution part of the inequality system, however, might select an execution in which a task waits to communicate with another task while other communication partners stand ready. For example, if I return to the example of Figure 6.4, but add an event e to customer 2 before state 6, then the bound obtained, assuming all events take unit time, will not be sharp. The execution part will select an execution in which the resource communicates with customer 2 first, producing a wait path of length three through events e, c, and b. Since customer 2 is not ready to communicate with the resource until one time unit after both the resource and customer 1 are ready, this violates the assumption of maximal parallelism.

6.2.3 Finding An Upper Bound on the Time Between Events

The technique presented above can easily be generalized to find an upper bound on the time between two specific events, as in the uniprocessor technique. To accomplish this, I modify each part of the inequality system as follows:

- For the execution part, I generate an inequality system that finds segments of an execution beginning and ending with specific events, as I did for the uniprocessor technique of Section 6.1.

- For the critical path part, I assign begin and halt variables differently. Flow enters the FSAs in the timed interval either through connection variables (if an initial interval is used) or through start variables (if no initial interval is used). I assign a begin variable to every state that has one of these connection or start variables. Flow exits the FSAs in the timed interval through a different set of connection variables and I add a halt variable to every state that has one of these connection variables.

- For the bounding part, I use the transition variables for the timed interval to bound the wait path variables.

Note that transitions removed from the FSAs by the marking algorithm can also be removed from the potential wait graph (along with any associated cross arcs). With these changes, the critical path part will find a wait path through the potential wait graph, from a state in which some FSA could be at the beginning of the timed interval to a state in which some FSA could be at the end of the timed interval, passing only through events that occur in the timed interval.

6.2.4 Tightening the Bound

The bound derived by the above technique may not be sharp for several reasons, as discussed in Section 6.2.2. Here, I discuss three methods for improving the bounds obtained. The first involves unrolling loops to eliminate cycles in the potential wait graph. The second involves finding another solution to the inequality system if the solution found does not yield a sharp bound. The last method involves splitting the timed interval into multiple intervals.

As mentioned in Section 6.2.2, cycles in the potential wait graph can allow spurious cyclic flows to degrade the bound obtained. I believe that the only solution to this problem is to remove as many cycles as possible from the potential wait graph by unrolling loops in the tasks. The uniprocessor technique of Section 6.1 can be used

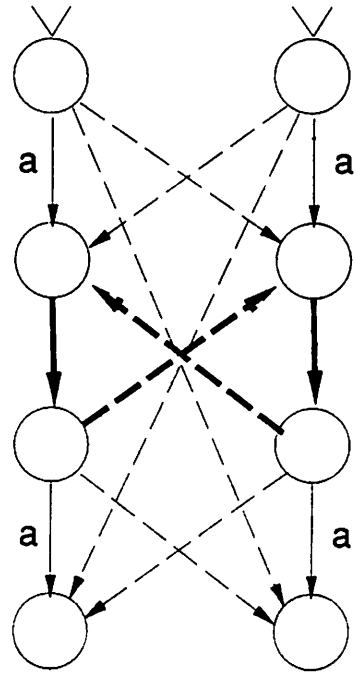


Figure 6.9. Example of Cycle in Potential Wait Graph Created by Cross Arcs

to obtain a bound (if one exists) on the number of times a loop may be executed in the timed interval. To accomplish this, I assign unit duration to one event within the loop that is always executed once per iteration, and I assign zero duration to all other events. The maximum value of the objective function is an upper bound on the number of iterations of the loop in any execution. Given bounds on all loops, it is always possible to unroll a task containing loops into another task that does not such that the possible traces of the tasks are the same. Such unrolling, however, can produce very large tasks when the loop bounds are not small or when the contents of the loop are more complicated (e.g., branches, nested loops, etc.).

Note that removing loops from the tasks does not remove all cycles from the potential wait graph. The cross arcs for communications that occur more than once can also form cycles, as shown in Figure 6.9 by the cycle in bold. Some of these cycles could be removed using information on which matchings of communication events are possible in executions.

The second method for improving the bounds builds on the first method and involves finding successively better bounds by excluding solutions to the inequality system that are not known to be sharp. If loops are eliminated from the tasks, then the integer programming variables are bounded above by one since no transition in a task FSA can be taken more than once. This bound holds even if cycles remain in the potential wait graph. Such a bound allows us to exclude one specific integral solution from an inequality system without removing any others. Therefore, if I obtain a bound that I cannot verify is sharp, I can add an inequality to the inequality system excluding that solution and rerun the integer programming package to obtain another solution. This can be repeated until the bound obtained is known to be sharp.

I proceed as follows. Suppose I have an inequality system with variables z_i, $i = 1, \ldots, n$ $(0 \leq z_i \leq 1)$ and objective function $\sum_{i=1}^{n} c_i z_i$ with $c_i > 0$ for all $i = 1, \ldots, n$. I will represent a solution S by the set of indices of variables that are one: $S = \{i \mid z_i = 1\}$. Given a solution, S, of this inequality system maximizing the objective function, I generate the additional inequality

$$\sum_{i \in S} c_i z_i < \sum_{i \in S} c_i$$

Theorem 6.3 *Given that S is an integral solution to a linear inequality system with variables $0 \leq z_i \leq 1$, $i = 1, \ldots, n$ maximizing the objective function $\sum_{i=1}^{n} c_i z_i$ $(c_i > 0)$, the added inequality*

$$\sum_{i \in S} c_i z_i < \sum_{i \in S} c_i$$

prohibits only solution S.

Proof. The added inequality clearly prohibits solution S. Suppose the added inequality prohibits another integral solution, S'. Define

$$p = \sum_{i \in S \cap S'} c_i$$

$$q = \sum_{i \in S} c_i$$

$$r = \sum_{i \in S'} c_i$$

Note that p and q are, respectively, the value of the left and right hand sides of the added inequality on solution S', while r is the value of the objective function at solution S'. There are three cases:

1. $r > q$: This contradicts the optimality of solution S.

2. $r < q$: Since $S \cap S' \subseteq S'$, I have $p \leq r$ which implies $p < q$, which contradicts S' being prohibited by the added inequality.

3. $r = q$: Since $S' \subset S$ would imply case two, there must exist a $j \in S' - S$ which implies $p < r$ and hence $p < q$. This contradicts S' being prohibited by the added inequality.

Thus there does not exist another solution S' prohibited by the added inequality. □

As mentioned above, this procedure can be iterated to eliminate solutions yielding bounds not known to be sharp. I can know that a solution gives a sharp bound as follows. First, I check to see that the solution to the execution part of the inequality system corresponds to a possible execution by constructing a trace. The behavior generator tool, described in Section 3.3.5, can do this automatically (although it has not yet been modified to interface with the new inequality generator). If the solution does correspond to an execution, I can build the wait graph for that execution, as described in Section 6.2.1, and label each action with the earliest time it could commence using the graph-based algorithm sketched in that section. From this I can determine if any tasks waited to communicate with another task while an alternative communication partner was ready (i.e., maximal parallelism was violated). If the execution is acceptable, then I compare the length of the critical path found through the wait graph by the graph-based algorithm to the length of the wait path found (i.e., the bound given by the solution). If these are equal, the bound must be

sharp. In all other cases—the solution does not correspond to an execution, the execution found by the solution violates the maximal parallelism assumption, or the length of the wait path found by the solution (including any spurious cyclic flows) is greater than the length of the critical path for that execution—the bound may not be sharp. Since there exists a solution corresponding to the critical path through the longest execution, I may discard other solutions by generating the additional inequality described above and then find another. The result of an experiment with part of this method is described in Section 7.2.5.

Unfortunately, this method does not seem to generalize to the case where the variables are not boolean. The inequality added by the above method to exclude a solution would also exclude solutions in which the same variables are contributing the same total weight to the objective function, but the weight has been "shifted around" among the variables (e.g., if variables z_1 and z_2 both have weight 1, then excluding solution $z_1 = 1,\ z_2 = 2$ would also exclude $z_1 = 2,\ z_2 = 1$). I know of no way to enumerate by cost the integral solutions of an arbitrary inequality system using an integer programming package.

The third method for tightening the bounds derived by this technique uses multiple timed intervals to represent the timed part of an execution. To prevent solutions containing spurious wait paths due to impossible communication event matchings, I might divide the timed part of the execution into different intervals at certain key events. I would generate a copy of the potential wait graph for each interval and connect these copies as follows: I add an arc out of each node in a copy of the potential wait graph to that same node in the copy for the next interval (if present). The bounding part bounds the critical path variables in a copy of the potential wait graph with the transition variables of its corresponding interval system. Note that cross arcs between communication events are generated only for matching events in the same interval. In Section 7.2.5, I will illustrate how this technique could be used to eliminate a spurious solution in the reactor monitor example.

6.3 Summary

The state explosion problem for concurrent systems with real-time requirements is often worse than for untimed systems since the presence of time, if included in the state of the system, can greatly increase the number of states. This chapter presents techniques for deriving bounds on the time that can elapse between events in a concurrent real-time system. Section 6.1 described a technique for deriving both upper and lower bounds on the time that can elapse between two specific events when the concurrent program is run in a uniprocessor setting (i.e., events do not overlap in time). This technique uses the technology developed for star-less queries, described in Section 5.1, and the integer programming package's ability to find a solution that optimizes a given objective function. The objective function is set equal to the sum of the durations of the events that occur between the given pair of delimiting events. Section 6.2 described a technique for deriving an upper bound on the time that can elapse between two specific events when the concurrent program is run in a maximally-parallel multiprocessor setting (i.e., each task has its own processor). This technique uses the concept of a critical path to find the parallel execution time of a set of events, and uses the uniprocessor technique to find sets of events that represent legal executions. Both techniques are based on the inequality-based constrained expression analysis technique and use integer programming to avoid the state explosion problem. The results of experiments with these techniques are presented in Chapter 7.

Chapter 7

Experimental Evaluation

This chapter describes a preliminary empirical evaluation of many of the techniques presented in Chapters 4, 5, and 6. Section 7.1 describes the implementation of a prototype tool automating the ω-star-less query technique of Section 5.3, the critical race technique of Section 5.5, the uniprocessor real-time bound technique of Section 6.1, and the multiprocessor real-time bound technique of Section 6.2. A series of experiments with these techniques that were conducted using this tool are described in Section 7.2, along with an additional experiment with the identical tasks and counter variable techniques of Chapter 4, which was conducted without the tool.

7.1 Implementation

This section describes the implementation of a prototype tool constructed to conduct the experiments presented in Section 7.2. Section 7.1.1 describes at a high level how the constrained expression toolset as a whole (presented in Section 3.3) was modified to conduct the experiments. The remaining sections describe optimizations for reducing the size of the generated inequality systems (in hopes of reducing the difficulty of the integer programming). Section 7.1.2 describes a graph marking algorithm that can reduce the size of the generated inequality system by removing parts of the FSAs that are unreachable in a given interval. Section 7.1.3 describes a technique that allows multiple intervals to share a single interval system. Finally, Section 7.1.4 describes several other optimizations.

7.1.1 *Modifications to the Toolset*

The constrained expression toolset described in [8] and Section 3.3 was modified to conduct most of the experiments in this chapter. Almost all of the modifications were in the inequality generator component, which was completely rewritten to support the generation of the many additional type of inequalities needed for the new analysis techniques. The new tool, described in detail in [22], is roughly 3000 lines of COMMON Lisp. It uses several layers of abstraction to improve efficiency, comprehensibility, and extensibility. At the highest level, queries formed from intervals and sequences are used to construct a *flowgraph* and a set of *symbolic inequalities* on the next level. These structures are then processed to produce the actual inequality system on the lowest level. The first transformation allows additional analysis techniques to be incorporated by simply describing the flowgraph and additional inequalities they create. The second transformation increases the efficiency of the generated inequality system by applying low level optimizations, sketched in Section 7.1.4.

The new inequality generator automates the following analysis techniques:

- The ω-star-less query technique of Section 5.3.
- The generation of fairness inequalities for multiple callers of an entry, described in Section 5.2.1, but not the technique for enforcing fairness in an Ada `select` statements, described in the same section.
- The basic critical race technique of Section 5.5.1, but not the multi-way race technique described in Section 5.5.2.
- The technique for deriving bounds on concurrent real-time systems run on a uniprocessor, described in Section 6.1, except for the periodic tasks technique in Section 6.1.4.
- The technique for deriving bounds on concurrent real-time systems run on a multiprocessor in a maximally-parallel fashion, described in Section 6.2, except

for the techniques to tighten the bound in Section 6.2.4. Of the three techniques presented in that section, only part of the second (finding another solution) has been implemented.

In the remainder of this section, I describe the modifications to the other tools that were made for these experiments.

The new inequality generator generates the equations for the tasks from FSAs only (not from REs or REDFAs). Various optimizations are used to reduce the size of the inequality system generated from the tasks without converting them to REs or REDFAs, as was done previously. Therefore, the constraint eliminator was modified to convert all task expressions to DFAs, regardless of whether they have intra-task constraints.

The only change in IMINOS made since the experiments described in [8] is the addition of an option to use semantic information output by the inequality generator to help select the variable on which to branch. This addition has generally improved the performance of IMINOS on the inequality systems produced, as discussed in [9], and so I use it in the experiments described in Sections 7.2.2 and 7.2.3. In those experiments, this option was combined with another option that minimizes the search through the branch-and-bound tree by stopping at the first integral solution found rather than continuing until the solution is known to be optimal. Since I seek optimal solutions to derive bounds on real-time systems, I do not use this option for the experiments described in Sections 7.2.4 and 7.2.5.

The behavior generator has not yet been updated to accept input from the new inequality generator and so it was not used in the experiments described in this document. Instead, the new inequality generator provides the analyst with a possible trace of each task individually, constructed from the solution. The analyst examines the individual traces of the tasks to determine whether they are consistent with one another and hence represent a trace of the concurrent system.

7.1.2 *Marking Algorithm*

This section discusses a graph marking algorithm that can reduce both the size of the inequality system generated for an analysis and the likelihood of a spurious solution due to extra cyclic flow (as discussed in Section 3.2). The idea behind the algorithm is that parts of an FSA can be determined to be unreachable in certain intervals using a kind of reachability analysis on a single FSA. For example, the only reachable parts of an FSA with both a and b events in an interval starting after an a and ending with a b are those parts lying on paths in the FSA between an a and a b. I need not generate transition variables and flow equations for those parts of the FSA not on a path between an a and a b. Conceptually, I remove these parts of the FSA before generating the variables and inequalities. Clearly this can reduce the size of the inequality system generated, and, if cycles in the FSA are either removed or broken, then I have reduced the likelihood of a spurious solution due to cyclic flows. Removing or breaking cycles to prevent spurious cyclic flows is most important for the techniques that derive bounds for real-time systems, discussed in Chapter 6, since a solution maximizing the amount of flow encourages extra cyclic flows to appear.

The marking algorithm is based on the observation that there are two conditions under which it is clear that a transition k cannot actually occur in an interval starting after an a (the *start event*) and ending at a b (the *halt event*), and hence can be removed:

1. The transition k is in one of the FSAs containing a or b and does not lie along any possible path starting after an a, ending with a b, or both. For example, the lower cycle on d in M_3 of Figure 7.1 does not lie along any path ending with b, and so cannot be reached in an interval starting after an a and ending at a b.

2. The transition k is in an FSA with a *required* event—an event that must happen in the interval—and every path from the required event to transition k passes through a *forbidden* event—an event that cannot happen. The halt event is

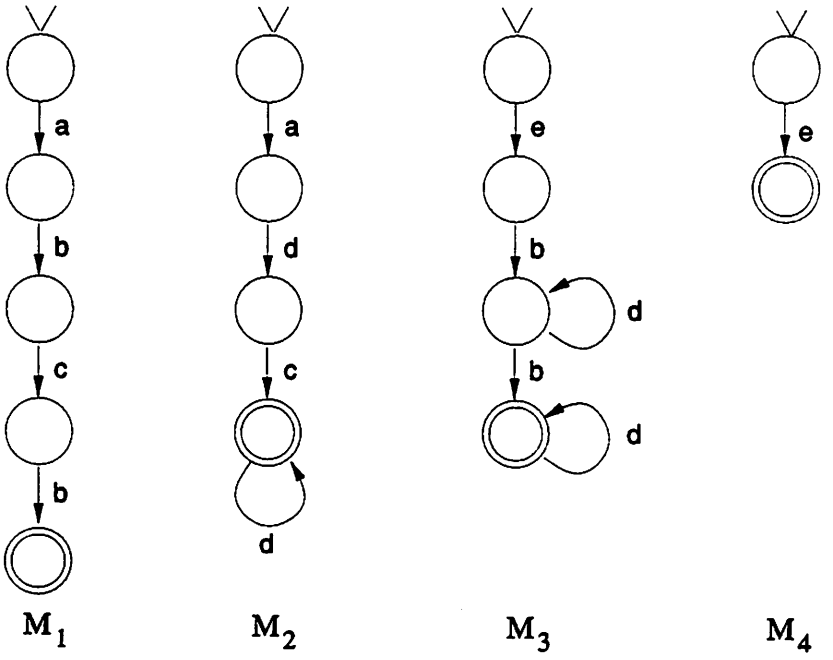


Figure 7.1. Example for Marking Algorithm

required, and the analyst may specify additional events as required or forbidden, as mentioned at the end of Section 5.1.1. The start event is not required in the sense of Section 5.1.1, since it does not actually occur in the interval, but I treat it as a required event in the context of this algorithm. A communication event in an FSA can also become forbidden if all of its matching events in other FSAs are removed. An example of this is given in Figure 7.1. The marking algorithm would remove the c transition in M_1 by the first condition (recall that, by the definition of a sequence query in Section 5.1.1, an interval starting after a and ending at b cannot contain any intervening a's or b's). This makes the c event in M_2 forbidden. I then know that the d cycle in M_2 cannot be reached since there is no path from the a event in that task, which must occur, to the cycle that does not pass through a forbidden event.

The marking algorithm is never applied to open intervals since the start and halt events in such intervals may occur more than once and so do not delineate the state of the automaton within the interval.

Input: An FSA M_j
A set of start events S such that $|S \cap alphabet(M_j)| \leq 1$
A set of halt events H such that $|H \cap alphabet(M_j)| \leq 1$
A set of required events R such that $R \supseteq S \cup H$
A set of forbidden events F
Output: FSA M_j with unreachable arcs removed
An updated set of forbidden events F

Mark all transition arcs $k \in trans(M_j)$ with the empty set $\emptyset$.
For each required event $e \in R$:
  If $e \notin H$ then
    For each transition $k \in trans(M_j)$ where $label(k) = e$:
      Search forward from k to all arcs reachable without
      crossing an arc labeled with a forbidden event, or
      continuing past an arc labeled with a halt event.
      Add e to the set marking each arc crossed.
  If $e \notin S$ then
    For each transition $k \in trans(M_j)$ where $label(k) = e$:
      Search backward from k to all arcs reachable without
      crossing an arc labeled with a forbidden event, or
      continuing past an arc labeled with a start event.
      Add e to the set marking each arc crossed.
Remove all transition arcs not marked with the set $alphabet(M_j) \cap R$.
For each channel c such that $caller(c) = M_j$ or $acceptor(c) = M_j$:
  If $\{k | k \in trans(M_j) \wedge label(k) \in symbols(c)\} = \emptyset$ then
    $F := F \cup symbols(c)$

Figure 7.2. Algorithm to Mark an FSA

The marking algorithm uses conditions 1 and 2 to remove as many transitions from the FSAs as possible. It accepts a set of start events, a set of halt events, a set of required events, and a set of forbidden events as input. The algorithm to mark one FSA, given these four sets, is shown in Figure 7.2. The algorithm updates the set of forbidden events with any (now) unmatched communication events. There should be at most one start event and at most one halt event per FSA, since it does not make sense for an FSA to begin or end the interval in more than one state.

Essentially, I mark only those arcs in the FSA that can be reached from all required events. Any remaining unmarked arcs cannot appear in any flow containing

all required events for that FSA and thus can be removed. The complete marking algorithm proceeds as follows. Each of the FSAs is placed on a list of FSAs to process. On each iteration, the first FSA on the list is removed and the procedure in Figure 7.2 is applied to this FSA. If an event is returned as forbidden by this procedure, and that event appears in an FSA, M_i, that is not still on the list, then more of the events in M_i may need to be removed, so M_i is put back on the list. The algorithm terminates when no FSAs remain on the list. Since both the time for each iteration and the number of iterations is linear in the size of the FSAs, the worst case running time of the marking algorithm is quadratic in the size of the FSAs.

As a heuristic for speeding the marking, I place the FSAs with the most start and halt events first on the list, since start and halt events have the greatest potential to mark parts of the FSA unreachable and cause events occurring only in those parts to be marked forbidden. In practice, using this heuristic, no FSA has ever been placed back on the list. For convenience, in the implementation, I limit the number of times an FSA can be placed back on the list to one.

Applying this algorithm to the example of Figure 7.1, I first place the FSAs on the list in the order $1, 2, 3, 4$. M_1 comes first because it has two distinct start/halt events (a and b). M_2 and M_3 come next since they have one start/halt event each. Finally, M_4 has no start/halt events. Applying the procedure of Figure 7.2 to M_1 will produce the marked FSA shown in Figure 7.3. Removing the two arcs not marked $\{a, b\}$ removes all c communication events in M_1, thereby making that event forbidden. In M_2, the search now stops at c, causing the d cycle to be removed. (Note that the presence of the first d in M_2 prevents d from being forbidden.) Next, in M_3, the lower d cycle is removed. The marking algorithm removes nothing from M_4. The final FSAs, from which inequalities would be generated, are shown in Figure 7.4.

For each interval, the marking algorithm is applied to the original FSAs and the inequalities for that interval are then generated from the pruned FSAs output by the marking algorithm. For a sequence query $e_1 \ldots e_n$, the first interval is given e_1 as the

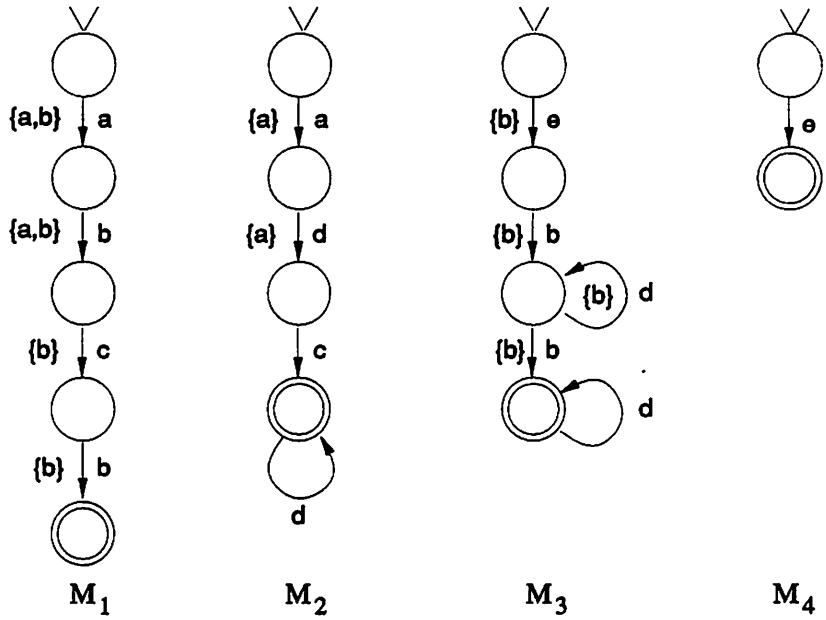


Figure 7.3. Markings of FSAs During Marking Algorithm

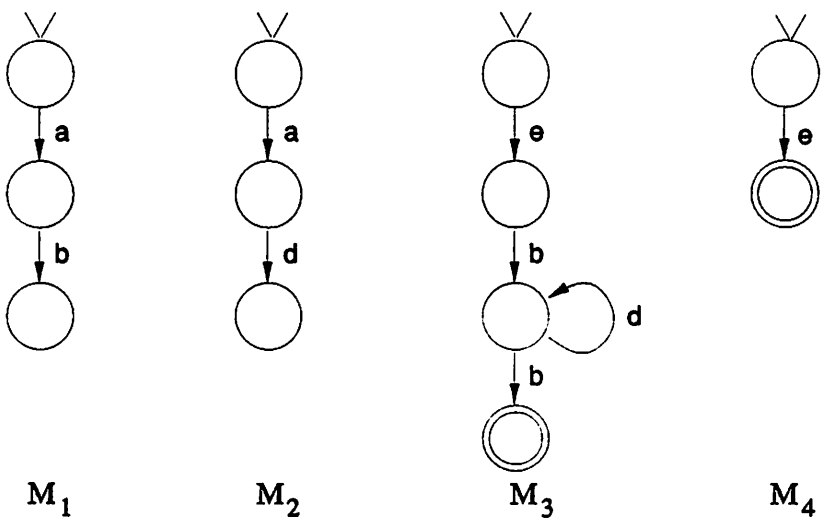


Figure 7.4. FSAs After Marking Algorithm

halt event and no start events, while interval $i = 2, \ldots, n$ is given e_{i-1} as the start event and e_i as the halt event. Given that Π is the alphabet of the query, the events in $\Pi - \{e_i\}$ are forbidden in interval i.

The marking algorithm is a standard part of the processing of an interval and is therefore used in the implementation of the ω-star-less query technique of Section 5.3, the critical race technique of Section 5.5, the uniprocessor bounds technique of Section 6.1, and the multiprocessor bounds technique of Section 6.2.

7.1.3 Sharing

The optimization technique described in this section involves using the same inequality system to find intervals in different sequences. Suppose I have the query $ab \cup cd$. The technique described in Section 5.1.2 generates an interval system for each of the four intervals in the query (a, b, c, and d). I can halve the size of the generated inequality system by using one interval system to find a segment of a trace ending with an a or a c, and another interval system to find an extension of that trace ending with a b or a d.

Suppose I am given n intervals, each from a different sequence. Let s_i be the sequence variable for the sequence containing interval i, and let e_i be the event ending interval i. I construct an interval system for an interval that will end with e_i exactly when s_i is one. To accomplish this, I generate an interval system as described in Section 5.1.1 except for the following:

- I assign a connection variable to a state if it would have been assigned one when generating the interval system for any of the intervals individually. In addition, if a particular interval system would have assigned a connection variable to a state, then I associate the sequence variable for that interval with the connection variable for that state. In other words, I record, for each connection variable, which sequences allow the flow to exit the interval from that state.

- For each connection variable, c, associated with sequence variables $s_{i_1}, \ldots, s_{i_m}$, I generate a *connection inequality* bounding the connection variable from above with the sum of its associated sequence variables: $c \leq \sum_{j=1}^{m} s_{i_j}$. Given that the sequence with variable s_i is being sought (i.e., $s_i = 1$), these inequalities will allow the flow to exit the interval at a state only if that state would have been assigned a connection variable in the interval system generated for interval i.

- The requirement inequalities for the different intervals are likely to conflict and must be changed. In the example above, the interval for a forbids c's and vice versa. In Section 5.1.2, I showed how to enforce inequalities requiring events to occur only if the sequence they are in is the sequence being sought. Here, I use a similar technique to enforce only the requirement inequalities for the sequence being sought. Given that event e is required to occur K_i times in interval i (K_i would be zero if e is forbidden in interval i), I add a requirement equation setting the number of occurrences of e in the interval system to $\sum_{i=1}^{n} K_i s_i$. This enforces all requirements on event e. If all of the intervals involved are open, then I instead require that the number of occurrences of e is greater than or equal to $\sum_{i=1}^{n} K_i s_i$. If some of the intervals are open and some are not, then I generate one interval system for all the open intervals and another for all the non-open intervals.

The resulting inequality system has the same solutions as the interval system ending with e_i when $s_i = 1$, but it is roughly the size of a single interval system. Perpetual intervals of different sequences may also share the same interval system using this technique. Thus an ω-star-less query can usually be answered with a number of interval systems equal to the length of the longest sequence in the disjunction. Since only the ω-star-less query technique uses multiple sequences, this optimization is only applied for that technique.

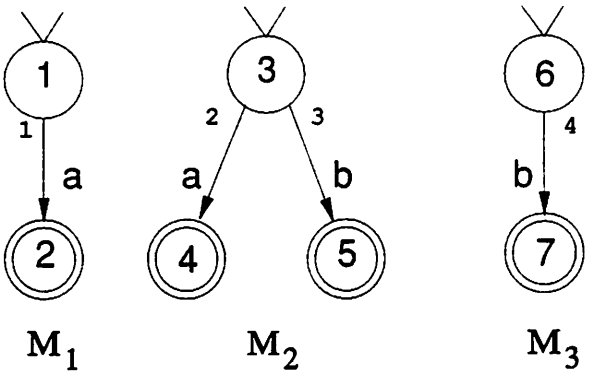


Figure 7.5. Example for Interval System Sharing

For an example of this optimization, consider the system in Figure 7.5. If I apply the star-less query $a \cup b$ to the system and use the above optimization, I obtain the inequality system in Figure 7.6. Since I am creating only one interval system, I omit the interval subscript on the transition and connection variables. Interval 1, ending with a, would assign connection variables to states 2, 4, 6, and 7. Interval 2, ending with b, would assign connection variables to states 1, 2, 5, and 7. I place a single connection variable on each of these states and generate a connection inequality for states 1, 4, 5, and 6 (which are assigned connection variables by only one interval) that prevents any flow through the connection variable unless the appropriate sequence is being sought. The requirement equations force one occurrence of a and zero occurrences of b if sequence 1 (containing interval 1) is sought, and forces one occurrence of b and zero occurrences of a if sequence 2 is sought

There are several restrictions on intervals that share an interval system. All of the intervals that share a given interval system must be initial (or not initial), final (or not final), open (or not open), perpetual (or not perpetual), and may only specify additional linear relations that are of a certain restricted form. The details of these restrictions are given in [22].

The marking algorithm is not applied to the FSAs of intervals sharing an interval system, since I cannot remove parts of an FSA that are unreachable in only some of the intervals. The marking algorithm could be augmented for this optimization by

Flow:			**(state)**
1	$=$	$x_1 + c_1$	(1)
x_1	$=$	c_2	(2)
1	$=$	$x_2 + x_3$	(3)
x_2	$=$	c_4	(4)
x_3	$=$	c_5	(5)
1	$=$	$x_4 + c_6$	(6)
x_4	$=$	c_7	(7)
Synchronization:			**(channel)**
x_1	$=$	x_2	(a)
x_3	$=$	x_4	(b)
Sequence Choice:			
$s_1 + s_2$	$=$	1	
Connection:			**(state)**
c_1	$\leq$	s_2	(1)
c_4	$\leq$	s_1	(4)
c_5	$\leq$	s_2	(5)
c_6	$\leq$	s_1	(6)
Requirement:			**(symbol)**
x_1	$=$	$1 \cdot s_1 + 0 \cdot s_2$	(a)
x_3	$=$	$0 \cdot s_1 + 1 \cdot s_2$	(b)

Figure 7.6. Inequality System for Interval Sharing

having it remove only parts of the FSAs that are unreachable in all intervals sharing an interval system. Since the marking algorithm is most important for the real-time techniques (described in Chapter 6), and these techniques do not use multiple sequences, I have chosen not to implement this extension to the marking algorithm at this time. I believe that not being able to use the marking algorithm for intervals sharing an interval system is the only disadvantage to this optimization, though I do not yet have enough experience with the techniques to gauge the magnitude of this disadvantage.

7.1.4 Other Optimizations

This section summarizes several other optimizations employed by the new inequality generator. The reader is referred to [22] for details. First, when assigning

connection variables to states in FSAs that do not contain the events marking the end of the interval, I need not assign a connection variable to every state. The FSAs generated by the toolset contain many symbols for events that can be considered to be internal (i.e., all events except those representing communication). I can reduce the number of connection variables by requiring that flow through an FSA in an interval end just after an external event. Since the synchronization equations are the only constraint inequalities involving event counts for individual intervals, it cannot matter at what state the flow changes intervals between one external event and the next. Thus I assign connection variables only to states having an incoming transition on an external event.

Other optimizations are also employed to reduce the size of the inequality system generated. The tool generates *symbolic variables* for the variables defined in this document (e.g., transition variables, connection variables, etc.) and *symbolic inequalities* in these variables. If a symbolic equation equates two symbolic variables, as would the flow equation for a state with one in transition and one out transition, then only one real ILP variable will be generated to represent the value of both symbolic variables and the equation will not be generated. This equating is transitive—any number of equated symbolic variables may be represented using a single real ILP variable. An inequality may also set a symbolic variable to zero. In this case, neither the zeroed symbolic variable nor any other symbolic variables it has been equated to will be generated. As a result of the equating and zeroing of symbolic variables, the number of real variables and inequalities generated is typically less than half of the number of symbolic variables and inequalities generated.

7.2 Experiments

This section describes a series of experiments with the analysis techniques presented in Chapters 4, 5, and 6. Section 7.2.1 describes an experiment with the identical tasks technique and the counter variable technique of Chapter 4. In Section 7.2.2, I

describe a series of experiments with the ω-star-less query technique of Section 5.3. Next, I present the results of a series of experiments with the techniques of Chapter 6 for deriving bounds for concurrent real-time systems. Sections 7.2.4 describes experiments with the uniprocessor technique of Section 6.1, while Section 7.2.5 describes experiments with the multiprocessor technique of Section 6.2.

7.2.1 Experiments with Identical Tasks and Counter Variables

This section describes an experiment in which both the identical tasks technique and the counter variable technique of Chapter 4 were applied together to analyze a small example. This experiment was conducted without the new inequality generator, which does not yet automate these techniques. I used the version of the toolset described in Section 3.3 and added some of the inequalities by hand. Completely automating these techniques will require modification of the deriver tool, which is old, brittle and slated for replacement. Due to the current lack of automated support, these techniques have not yet been tried on other examples.

I have done a limited empirical validation of the identical tasks and counter variable techniques by conducting an experiment with the toolset using the coupled resource allocator in Figure 4.7 (page 63). The CEDL code for the example is given in Figure 7.7 and differs slightly from the Ada code for this system given in Figure 4.6 (page 62). As I mentioned before, the toolset does not support the techniques described in Chapter 4, so a few words about how I conducted the experiment are in order. Counter variables were represented using an enumerated type in CEDL having the two values `zero` and `max`. The CEDL primitives `SUCC` and `PRED` were used as increment and decrement statements, since this causes the deriver to automatically generate the alternatives for abnormal termination due to overflow or underflow; the overflow alternative contains the event symbol `use(count;max)` while the underflow alternative contains the event symbol `use(count;zero)` where `count` is the name of the counter variable. The deriver also automatically generates

the appropriate hang alternatives given the guards of a select statement, labeling them with `use` symbols indicating the acceptable values of the variable for permanent blockage on the entry to occur. The deriver does not actually output *increment* and *decrement* symbols, but rather the sequences `use(count;zero) def(count;max)` and `use(count;max) def(count;zero)`. I treated these sequences as if they were *increment* and *decrement* symbols (i.e., to calculate the value of the counter variable, I subtract the number of occurrences of the latter sequence from the number of occurrences of the former).

The regular expressions output by the deriver were used to generate a set of inequalities (the derivation of inequalities from regular expressions is discussed in [8]). These regular expressions would normally be converted to automata by the constraint eliminator before inequality generation, but since I wanted to bypass that tool to use the counter variable technique, it was more convenient to generate the inequalities from the regular expressions. The inequalities generated from the regular expressions were then modified (by hand) as follows:

- The top level variable for the customer task expression (essentially equivalent to the implicit flow into the start state of its FSA) was set to R rather than one, and the hang inequalities for the entry calls involving this task were changed as described in Section 4.1. This creates R identical copies of the customer task.

- The range and end condition inequalities were added. Rather than adding the counter equations and variables, I simply substituted the expression for the counter's value (i.e., the initial value plus the sum of the transition variables representing increments minus the sum of the transition variables representing decrements) into the inequalities. Thus no ILP variables were actually created to represent the counter values.

The resulting inequality system was fed to the ILP package and any solution found was inspected to see if it corresponded to a trace of the concurrent system.

```
task body ALLOC1 is
type COUNTER is (zero,max);
COUNT1 : COUNTER := max;
begin
  loop
    select
      when (COUNT1 > zero) =>
        accept ACQUIRE1;
        COUNT1 := COUNTER'PRED(COUNT1);
    or
        accept RELEASE1;
        COUNT1 := COUNTER'SUCC(COUNT1);
    end select;
  end loop;
end ALLOC1;

task body ALLOC2 is
type COUNTER is (zero,max);
COUNT2 : COUNTER := max;
begin
  loop
    select
        accept ACQUIRE2;
        COUNT2 := COUNTER'PRED(COUNT2);
    or
        accept RELEASE2;
        COUNT2 := COUNTER'SUCC(COUNT2);
    end select;
  end loop;
end ALLOC2;

task body CUSTOMER is
begin
  loop
    ALLOC1.ACQUIRE1;
    ALLOC2.ACQUIRE2;
    ALLOC2.RELEASE2;
    ALLOC1.RELEASE1;
  end loop;
end CUSTOMER;
```

Figure 7.7. CEDL for Coupled Resource Allocator Example

Table 7.1. Toolset Performance on Coupled Resource Allocator

R	n_1	n_2	Tasks	Deriv	Ineq	IMINOS	Total	Size
500	490	490	502	25	3	2	30	36 × 39
500	490	489	502	25	3	2	30	36 × 39
1000	990	990	1002	25	3	2	30	36 × 39
1000	990	989	1002	25	3	2	30	36 × 39

Table 7.1 shows the performance of the original toolset when applied to this example. The table shows the number of customer tasks (R), the amount of the first resource originally available (n_1), the amount of the second resource originally available (n_2), the number of tasks in the system, and the times used by the components of the toolset in seconds on a DECstation 3100. In this analysis, I searched for finite traces of the concurrent system. The first two lines of the table give the results for systems with 500 customers; the first line shows a correct system and the second shows one with fewer units of the second resource, leading to an error. The third and fourth lines give the results for similar systems with 1000 customer tasks. Because the tasks contain only counter variables, it is not necessary to use the constraint eliminator in these analyses. IMINOS found solutions only for the two systems where there are fewer units of the second resource and these solutions correspond to finite traces in which allocator 2 aborts. The lack of integral solutions in the two systems with equal amounts of both resources proves that the tasks in these systems cannot become permanently blocked or abort. Note that the systems of inequalities are the same size and the execution times are the same for all versions of the system. The system with 1000 customers has at least 10^{596} reachable states.

One practical concern with both the identical tasks technique and the counter variable technique is the large coefficients they introduce into the inequality system and the potential numerical stability problems these can engender. In the above experiment, I tried larger values for R but found that the technique failed before R reached 1500. With coefficients this large, the integer programming package was

unable to find integral solutions I knew existed. By lowering the tolerance that defines how close a variable has to be to an integer before it is considered integral, I was able to analyze somewhat larger systems, but increasing this tolerance too much yielded integral solutions in systems I knew had none. In practice, an analyst would have to have some way to tell whether numerical instability had made the output of the integer programming package unreliable. Better performance might be achieved using other methods for ILP.

This experiment has shown the feasibility of using the techniques of Chapter 4 to analyze a large concurrent system whose state explosion is due to many identical components and counter variables having large ranges. More experiments with these techniques are planned once the deriver component of the toolset is reimplemented for easier extensibility. Specifically, the counter variable technique could be applied to the dining philosophers with host example described in [8]. In that example, the size of the host task's automaton grows quadratically with the number of philosophers it must count, greatly increasing the size and difficulty of the integer programming problems that must be solved. With the counter variable technique, this automaton would grow linearly and produce much smaller inequality systems. Also, both the identical tasks technique and the counter variable technique could be applied to the readers and writers problem, also described in [8], to analyze versions of this system with large numbers of readers and writers.

7.2.2 Experiments with ω-star-less Queries

This section describes a series of experiments that demonstrate the feasibility of answering ω-star-less queries with the technique of Section 5.3. I used four variants of a coterie mutual exclusion system. A coterie is a general mechanism for achieving mutually exclusive access to a resource in a distributed system [34]. Each resource has a set of keys. A *coterie* is a set of subsets of the keys with the property that any two subsets have a non-null intersection. A customer wishing to use the resource

must first acquire all of the keys in one of the coterie subsets. Since any two coterie subsets must share at least one key, only one customer may possess all of the keys in one of the coterie subsets at any given time. One simple type of coterie consists of all subsets possessing a majority of the keys.

Two properties of mutual exclusion protocols are commonly verified. First, the protocol must enforce the mutual exclusion. Second, the protocol should be "fair" in some sense—a customer wishing to use the resource should eventually be permitted to do so. The first property is a safety property; the second is (usually) a liveness property. I attempted to verify these properties with the toolset incorporating the new inequality generator, which automates the technique for ω-star-less queries described in Section 5.3. To verify that a system has a property P, I show that there does not exist a trace having the property $\neg P$. When I feed the toolset an ω-star-less query for $\neg P$, one of four things can happen:

- The toolset reports that the system of inequalities generated has no integral solution and thus that no trace matching the query exists. The system has been shown to have property P.

- The toolset finds a solution to the system of inequalities generated and the analyst verifies[1] that the solution does correspond to a trace with property $\neg P$. The system has been shown not to have property P.

- The toolset finds a solution to the system of inequalities generated, but the analyst discovers[2] that the solution does not correspond to a trace. In this case, the analysis is inconclusive.

- The integer programming package is unable to either find an integral solution or prove that none exists within a preset bound on the number of branch-and-bound

[1]If the behavior generator described in Section 3.3.5 were modified to interface with the new inequality generator, this verification could be done automatically by that tool.

[2]Again, a modified behavior generator could determine this automatically.

iterations (typically allowing the ILP package to run for about half an hour). In this case also, the analysis is inconclusive.

Each experiment is described by a precise statement of the property to be verified, a discussion of the formulation of this property as an ω-star-less query, and the result of attempting to answer this query with the toolset. Rather than litter the text with numbers, the information on toolset performance is collected in Table 7.2 on page 185. It lists, for each example and query: the result of the experiment, the time (in seconds on a DECstation 5000/125) taken by each of the tools, the total time, and the size of the inequality system generated (number of inequalities $\times$ number of variables). The query names match those in the figures showing the actual queries fed to the toolset, or those mentioned in the description of the experiment. I also analyzed incorrect versions of the systems to show that the necessary conditions generated are useful for finding errors when they are present.

I considered four versions of a coterie mutual exclusion system containing one resource, three guards, and either two or three customers. The customers must acquire a majority of the keys (i.e., two keys) to use the resource. Each guard holds one key and a customer communicates with a guard to request, acquire, and release its key.

In the first version, there are three customers and three guards. Each customer requests a key from two specific guards. If the guards and customers are pictured alternating in a circle, then each customer requests a key from the guards immediately beside it. This communication pattern is similar to the dining philosophers system, where customers take the place of philosophers and guards take the place of forks. A customer requests and is granted a key by communicating with a guard at a `REQUEST` entry (i.e., once this communication completes, the customer has the key). A customer releases a key by communicating with the key's guard at the guard's `FREE` entry. The guards are ordered and the customers must request keys in this order—this standard technique for deadlock prevention stops the circular deadlock present in the dining philosophers system from occurring here. Once exclusive access to the resource has

```
task body customer_a is
begin
  loop
    guard_1.REQUEST_1;
    guard_2.REQUEST_2;
    resource.ENTER;
    resource.LEAVE;
    guard_1.FREE_1;
    guard_2.FREE_2;
  end loop;
end customer_a;
```

```
task body guard_1 is
begin
  loop
    accept REQUEST_1;
    accept FREE_1;
  end loop;
end guard_1;
```

Figure 7.8. CEDL for Customer and Guard in Version One

been obtained, a customer communicates with the resource at entries `ENTER` and `LEAVE` to represent the beginning and end of the customer's use of the resource. The CEDL source for one customer and one guard is shown in Figure 7.8. The customers are labeled a, b, and c and the keys are labeled 1, 2, and 3.

The first property I attempted to verify is mutual exclusion: a customer cannot begin to use the resource while another customer is using it. There are six possible ways that this property can be violated, depending on which customers are involved (e.g., customer a begins to use the resource while customer b is using it, customer b begins while customer a is using it, etc.). I can express each particular violation, say customer b interrupts customer a, with an ω-star-less query having one sequence of two intervals. The first interval is initial, open, and ends with customer a beginning to use the resource. The second interval ends with customer b beginning to use the resource and forbids customer a from finishing with the resource. Any possible violation of the property is represented by the ω-star-less query consisting of the six sequences expressing the six ways the property could be violated. The actual query I fed to the toolset is shown in Figure 7.9.

The Lisp-like syntax of the query is for the convenience of the inequality generator and is summarized in Appendix B. The arguments of `defquery` are: the name of the query, the string `"fair"` or `"nofair"` (indicating whether fairness inequalities will be generated, as described in Section 5.2.1), and the query itself. An ω-star-less query

```
(defquery "mutex" "nofair" (omega-star-less
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_a;resource.enter")))
    (interval :ends-with '((rend "customer_b;resource.enter"))
              :forbid '((rend "customer_a;resource.leave"))))
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_b;resource.enter")))
    (interval :ends-with '((rend "customer_a;resource.enter"))
              :forbid '((rend "customer_b;resource.leave"))))
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_a;resource.enter")))
    (interval :ends-with '((rend "customer_c;resource.enter"))
              :forbid '((rend "customer_a;resource.leave"))))
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_c;resource.enter")))
    (interval :ends-with '((rend "customer_a;resource.enter"))
              :forbid '((rend "customer_c;resource.leave"))))
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_c;resource.enter")))
    (interval :ends-with '((rend "customer_b;resource.enter"))
              :forbid '((rend "customer_c;resource.leave"))))
  (sequence
    (interval :initial t :open t
              :ends-with '((rend "customer_b;resource.enter")))
    (interval :ends-with '((rend "customer_c;resource.enter"))
              :forbid '((rend "customer_b;resource.leave"))))))
```

Figure 7.9. Query for Violation of Mutual Exclusion in Version One

takes as arguments a list of sequence queries (conceptually disjoined). Each sequence query takes as arguments a sequence of interval queries. The arguments to an interval query specify the parameters described in Section 5.3. When rendezvous are referred to in these queries, the **rend** macro is used. This macro expands the name of the rendezvous into the appropriate event symbols given the context.

When the query in Figure 7.9 was fed to the toolset, it reported that no trace matching the query exists, thus proving that the system enforces mutual exclusion. I modified the specification for customer *a* so that it does not request key 2 before using the resource. When the same query was applied to this incorrect system, the toolset found a solution corresponding to a prefix of a trace matching the query in which customer *a* obtains key 1 and begins to use the resource, then customer *b* obtains keys 2 and 3 and begins to use the resource.

The second property I attempted to verify is the fairness of the guards in disbursing keys. Specifically, I want to know if a customer can become permanently blocked while requesting a key. Having demonstrated the use of multiple sequences in an ω-star-less query, I shall pick a specific customer and key for these queries in order to focus on the construction of a sequence matching one possible violation; verification of the general properties would require multiple sequences (as in the mutual exclusion query described above), multiple runs of the analysis to test the possible cases, or an appeal to the symmetry of the system. In the first query, shown in Figure 7.10, I tested for the existence of a finite trace in which customer *a* becomes permanently blocked while requesting key 1. The toolset reported that no such trace exists. This proves that there is no deadlock involving all the tasks in which customer *a* becomes blocked while requesting key 1. This is the exactly the kind of analysis the original analysis technique could perform: a query consisting of a single finite interval.

Unfortunately, this analysis says nothing about whether customer *a* can become permanently blocked in an infinite trace—one in which the other tasks continue to execute forever. This type of blockage is usually called *starvation*. The ω-star-less

```
(defquery "deadlock" "nofair" (omega-star-less (sequence
  (interval :initial t :final t
            :require '("hang_c(customer_a;guard_1.request_1)")))))
```

Figure 7.10. Query for Deadlock in Version One

```
(defquery "starve" "nofair" (omega-star-less (sequence
  (interval :initial t
            :require '("hang_c(customer_a;guard_1.request_1)"))
  (interval :perpetual t))))
```

Figure 7.11. Query for Starvation in Version One (No Fairness)

query to test for the starvation of customer *a* requesting key 1 is shown in Figure 7.11. When this query was applied to the system, the toolset found a solution corresponding to a trace in which customer *a* becomes permanently blocked while requesting key 1; customer *b* repeatedly acquires keys 2 and 3, uses the resource, and releases the keys; and customer *c* acquires key 1 but then becomes permanently blocked while requesting key 3. Such an execution is possible only if no fairness is enforced at the REQUEST entries. If fairness is enforced for multiple callers of an entry (as described in Section 5.2.1), then customer *c* could not become permanently blocked while requesting key 3 since customer *b* completes a rendezvous at the same entry infinitely often. The query in Figure 7.12 instructs the toolset to generate the additional inequalities that enforce fairness at all entries. When this query was applied to the system, the toolset reported that there are no traces matching the query. This proves that customer *a* cannot become permanently blocked while requesting key 1 given that callers of an entry are selected fairly (i.e., a caller cannot be passed over forever). I modified the system so that the circular deadlock present in the dining philosophers system could occur. When this same query is applied to the incorrect system, the toolset found a solution corresponding to a finite trace containing the deadlock in which customer *a* becomes permanently blocked while requesting key 1.

```
(defquery "starve-fair" "fair" (omega-star-less (sequence
  (interval :initial t
            :require '("hang_c(customer_a;guard_1.request_1)"))
  (interval :perpetual t))))
```

Figure 7.12. Query for Starvation in Version One (Fairness Enforced)

Note that **hang** symbols in the queries of Figures 7.11 and 7.12 are required to be present in the interval rather than at the end of the interval. Surprisingly, if the first interval of the query in Figure 7.11 is specified as ending with the **hang** symbol, then the toolset will report no trace matching the query exists. The toolset is technically correct; this turns out to be an ill-formed query. The reason lies in the FSA representation of a CEDL task. A **hang** symbol is always followed by a **stop** symbol before the FSA enters an accepting state. Since this **stop** symbol is not in an SCC, it cannot occur in the perpetual interval. By ending the finite interval at the **hang** symbol, any trace containing that symbol is partitioned in such a way that it is impossible to follow that symbol with only events that occur infinitely often, since the **stop** symbol must occur before customer *a* can terminate. Although an experienced analyst would avoid this, and the tools themselves could check for this kind of anomaly in a query, the difficulty of formulating correct queries can be considered a weakness of the technique. I will return to this in Chapter 8.

The second version of the coterie mutual exclusion system I analyzed is similar to the first except that a strong type of fairness is built into the guards. Again, there are three customers, each of which tries to acquire two specific keys. In this version, however, the requesting and granting of keys are done with separate communications. To acquire a key, a customer must first communicate with a guard at the guard's **REQUEST** entry. The customer then waits for the guard to respond at the customer's **GRANT** entry. Only when this second communication completes does the customer have the key. Guards accept requests for keys at any time (unlike in version 1, where a guard would only accept requests it could immediately grant). Upon receipt

```
task body customer_a is
begin
  loop
    guard_1.REQUEST_1a;
    accept GRANT_1a;
    guard_2.REQUEST_2a;
    accept GRANT_2a;
    resource.ENTER;
    resource.LEAVE;
    guard_1.FREE_1a;
    guard_2.FREE_2a;
  end loop;
end customer_a;
```

Figure 7.13. CEDL for Customer in Version Two

of a request, the guard grants the key to the requestor immediately if the key is free. Otherwise, the guard waits until the key is released and then grants it to the requestor. This insures that once a request is made, it cannot be ignored indefinitely. The CEDL specifications of a customer and guard task for this version are shown in Figures 7.13 and 7.14 respectively. In the guard task, the variables `lock_a` and `lock_c` have the value true when the key is held by customers a and c respectively. The variable `waiting` is true if the customer not holding the key is currently waiting for it. The guards in the `select` statement of Figure 7.14 are logically unnecessary, but they reduce the size of the automaton produced.

For this version, I attempted to verify mutual exclusion using a query like the one of Figure 7.9. Here I introduced a small optimization to reduce the size of the inequality system that can often be used when seeking a prefix of a trace. Since all of the **hang** symbols in a trace can be forced to occur at the end, after the prefix sought, these symbols can be forbidden from the prefix as long as the query does not involve **hang** symbols. The only difference between the query used to verify mutual exclusion in this version and the query of Figure 7.9 is that I forbid all **hang** symbols in all intervals of the query. The toolset reported no trace matching this query exists, thus proving that mutual exclusion is enforced. I created an incorrect version of the

```
task body guard_1 is
waiting, lock_a, lock_c : boolean := false;
begin
  loop
    select
      when not lock_a and not waiting => accept REQUEST_1a;
      if lock_c then
        waiting := true;
      else
        customer_a.GRANT_1a;
        lock_a := true;
      end if;
    or
      when not lock_c and not waiting => accept REQUEST_1c;
      if lock_a then
        waiting := true;
      else
        customer_c.GRANT_1c;
        lock_c := true;
      end if;
    or
      when lock_a => accept FREE_1a;
      lock_a := false;
      if waiting then
        customer_c.GRANT_1c;
        lock_c := true;
        waiting := false;
      end if;
    or
      when lock_c => accept FREE_1c;
      lock_c := false;
      if waiting then
        customer_a.GRANT_1a;
        lock_a := true;
        waiting := false;
      end if;
    end select;
  end loop;
end guard_1;
```

Figure 7.14. CEDL for Guard in Version Two

system in which guard 2 will grant a key to customer *b* at any time (i.e., even when customer *a* has the key). Applying the query to the incorrect system produced a solution corresponding to a prefix of a trace in which customer *a* acquires keys 1 and 2 and begins to use the resource and then customer *b* acquires keys 2 and 3 and begins to use the resource.

The fairness property for the second version is a bit different than the fairness property for the first. In the second version, the guards should enforce the queuing of requests for the keys—a very strong kind of fairness that is actually a safety property. As with the mutual exclusion property, there are many possible violations with different participants. I picked one such violation and verified it could not occur. The query for one possible violation is shown in Figure 7.15. This query is satisfied by a trace in which customer *a* requests key 1, customer *c* requests key 1, but then customer *c* is granted key 1 before customer *a*. Note the use of the macro `prefix` to forbid all the **hang** symbols; this macro expands to a list of all event symbols having the indicated prefix. When fed this query, the toolset reported no trace matching the query exists, proving that this violation of the fairness property is impossible. I modified the system so that guard 1 ignores requests by customer *a* if customer *c* has the key (i.e., the request will be accepted, but the guard will not set the `waiting` flag and so will not grant the key to customer *a* when customer *c* returns it). When the query of Figure 7.15 was applied to this incorrect system, the toolset found a solution corresponding to a prefix of a trace in which customer *c* requests and is granted key 1, customer *a* requests key 1, customer *c* releases key 1 and requests it again, and finally customer *c* is granted key 1.

The third version of the coterie mutual exclusion system is significantly different from the first two versions in that there are only two customers but they will each try to acquire keys from any of the guards. A customer requests a key from a guard by communicating at the guard's **REQUEST** entry. The guard then immediately responds at the customer's **GRANT** entry, if the key is free, otherwise it immediately responds at

```
(defquery "queue" "nofair" (omega-star-less (sequence
  (interval :initial t  :open t :forbid '((prefix "hang"))
            :ends-with '((rend "customer_a;guard_1.request_1a")))
  (interval :forbid '((prefix "hang")
                      (rend "guard_1;customer_a.grant_1a"))
            :ends-with '((rend "customer_c;guard_1.request_1c")))
  (interval :forbid '((prefix "hang")
                      (rend "guard_1;customer_a.grant_1a"))
            :ends-with '((rend "guard_1;customer_c.grant_1c"))))))
```

Figure 7.15. Query for Queuing Violation in Version Two

the customer's `NO` entry, indicating that the key is unavailable. If a key is unavailable, the customer will request a key from a different guard. Each customer cyclicly requests keys from the guards whose keys it does not hold until it holds two keys. For example, at first, a customer would request keys from guard 1, then 2, then 3, then 1, etc. until it is granted a key, say key 2. The customer would then make requests to guards 1 and 3 until one of these guards grants its key. The CEDL specifications of a customer and guard task for this version are shown in Figures 7.16 and 7.17 respectively. In the customer task, the `lock` variables represent which keys are currently held by the customer. In the guard task, the `lock` variables represent which customer is currently holding the key.

For the third version, I tried to verify the property of mutual exclusion with the query in Figure 7.18. When this query was fed to the toolset, the integer programming package was unable to either find an integral solution or prove that none exists within the preset bound on the number of branch-and-bound iterations. The analysis of this query was therefore inconclusive. I simplified the query by splitting it into two queries each containing only one sequence. I selected one of these, shown in Figure 7.19, for analysis; to verify full mutual exclusion, I would have had to analyze the other query as well. The toolset reported that no trace matching this query exists. I modified the system so that customer *b* will assume it has key 3 even after receiving a refusal from guard 3. When the original two sequence query was applied to the incorrect version,

```
task body customer_a is
lock1, lock2, lock3 : boolean := false;
begin
  loop
    while not ((lock1 and lock2) or (lock1 and lock3) or (lock2 and lock3)) loop
      if not lock1 and not (lock2 and lock3) then
        guard_1.REQUEST_1a;
        select
          accept GRANT_1a; lock1 := true;
        or
          accept NO_1a;
        end select;
      end if;
      if not lock2 and not (lock1 and lock3) then
        guard_2.REQUEST_2a;
        select
          accept GRANT_2a; lock2 := true;
        or
          accept NO_2a;
        end select;
      end if;
      if not lock3 and not (lock1 and lock2) then
        guard_3.REQUEST_3a;
        select
          accept GRANT_3a; lock3 := true;
        or
          accept NO_3a;
        end select;
      end if;
    end loop;
    resource.ENTER;
    resource.LEAVE;
    if lock1 then
      guard_1.FREE_1a; lock1 := false;
    end if;
    if lock2 then
      guard_2.FREE_2a; lock2 := false;
    end if;
    if lock3 then
      guard_3.FREE_3a; lock3 := false;
    end if;
  end loop;
end customer_a;
```

Figure 7.16. CEDL for Customer in Version Three

```
task body guard_1 is
lock_a, lock_b : boolean := false;
begin
  loop
    select
      when not lock_a => accept REQUEST_1a;
      if lock_b then
        customer_a.NO_1a;
      else
        customer_a.GRANT_1a;
        lock_a := true;
      end if;
    or
      when not lock_b => accept REQUEST_1b;
      if lock_a then
        customer_b.NO_1b;
      else
        customer_b.GRANT_1b;
        lock_b := true;
      end if;
    or
      when lock_a => accept FREE_1a;
      lock_a := false;
    or
      when lock_b => accept FREE_1b;
      lock_b := false;
    end select;
  end loop;
end guard_1;
```

Figure 7.17. CEDL for Guard in Version Three

```
(defquery "mutex" "nofair" (omega-star-less
  (sequence
    (interval :initial t :open t :forbid '((prefix "hang"))
              :ends-with '((rend "customer_a;resource.enter")))
    (interval :ends-with '((rend "customer_b;resource.enter"))
              :forbid '((rend "customer_a;resource.leave")
                        (prefix "hang"))))
  (sequence
    (interval :initial t :open t :forbid '((prefix "hang"))
              :ends-with '((rend "customer_b;resource.enter")))
    (interval :ends-with '((rend "customer_a;resource.enter"))
              :forbid '((rend "customer_b;resource.leave")
                        (prefix "hang"))))))
```

Figure 7.18. Query for Violation of Mutual Exclusion in Version Three

```
(defquery "mutex-1" "nofair" (omega-star-less (sequence
  (interval :initial t :open t :forbid '((prefix "hang"))
            :ends-with '((rend "customer_a;resource.enter")))
  (interval :ends-with '((rend "customer_b;resource.enter"))
            :forbid '((rend "customer_a;resource.leave")
                      (prefix "hang"))))))
```

Figure 7.19. Simplified Query for Violation of Mutual Exclusion in Version Three

the toolset found a solution corresponding to a prefix of a trace in which customer a acquires keys 1 and 3 and starts using the resource while customer b acquires key 2, requests key 3 (but does not receive it), and then also starts using the resource.

The fairness property I attempted to verify for the third version is that a customer cannot be prevented from using the resource while the other customer uses it repeatedly forever. There are two ways this property can be violated, depending on which customer is starved. The case where customer a starves while customer b repeatedly uses the resource is expressed by the query in Figure 7.20. This query utilizes another type of interval specifier that allows a linear constraint to be added to the interval system. In this query, I am only interested in traces in which customer a repeatedly tries to acquire keys, so I require at least one of its requests to occur perpetually using the inequality in the `:restrict` parameter. Without this inequality

```
(defquery "starve" "nofair" (omega-star-less (sequence
  (interval :initial t :open t)
  (interval :perpetual t
            :restrict
              '((>= (total "beg_rend(customer_a;guard_1.request_1a)"
                           "beg_rend(customer_a;guard_2.request_2a)"
                           "beg_rend(customer_a;guard_3.request_3a)")
                    1))
            :require '((rend "customer_b;resource.enter"))
            :forbid '((rend "customer_a;resource.enter"))))))
```

Figure 7.20. Query for Starvation in Version Three

or some type of fairness in the `select` statement of the guards, customer a's call on a guard's `REQUEST` entry could be ignored forever. For this query, the toolset found a solution corresponding to a trace in which customer b repeatedly acquires keys 1 and 2, uses the resource, and releases the keys, while customer a acquires key 3 but is then perpetually denied keys 1 and 2. This proves that since the guards enforce no type of fairness when granting the keys, an unlucky customer can be denied access to the resource forever.

The fourth version of the coterie mutual exclusion system is a variant of the third version in which some fairness is built into the guards. In this version, if a guard must decline a request for a key, then once that key is released, the guard will not grant the key to the customer that last acquired it until it has been granted to some other customer (*the* other customer in this two customer version). The CEDL specification of the guard task for this version is shown in Figure 7.21 (the specification for the customer tasks is the same as in version three). In the guard task, the `lock` variables represent which customer is currently holding the key. The `wait` variable for a customer is true if that customer was the last to hold the key and should be forced to wait until the other customer acquires the key before being allowed to acquire the key again.

For this version, I first verified part of the mutual exclusion property (where customer b interrupts customer a), with the query in Figure 7.19. The toolset reports

```
task body guard_1 is
lock_a, lock_b, wait_a, wait_b : boolean := false;
begin
  loop
    select
      when not lock_a => accept REQUEST_1a;
      if lock_b then
        customer_a.NO_1a;
        wait_b := true;
      elsif wait_a then
        customer_a.NO_1a;
      else
        customer_a.GRANT_1a;
        lock_a := true;
        wait_b := false;
      end if;
    or
      when not lock_b => accept REQUEST_1b;
      if lock_a then
        customer_b.NO_1b;
        wait_a := true;
      elsif wait_b then
        customer_b.NO_1b;
      else
        customer_b.GRANT_1b;
        lock_b := true;
        wait_a := false;
      end if;
    or
      when lock_a => accept FREE_1a;
      lock_a := false;
    or
      when lock_b => accept FREE_1b;
      lock_b := false;
    end select;
  end loop;
end guard_1;
```

Figure 7.21. CEDL for Guard in Version Four

no trace matching this query exists. I modified the system as before so that customer b assumes it has key 3 after being refused it. Applying this query to the incorrect system yielded a solution corresponding to a prefix of a trace in which customer a acquires keys 1 and 3 and starts using the resource while customer b acquires key 2, requests key 3 (but does not receive it), and starts using the resource.

The modifications to the guard tasks in this version should prevent the possible starvation of a customer present in the third version. When the query of Figure 7.20 was applied to this system, the toolset produced a solution that does not correspond to a trace of the system due to spurious cyclic flows. The analysis of this query was therefore inconclusive. As with the mutual exclusion query in version 3, I broke the query down into cases to make it simpler for the analysis technique. This query has only one sequence, so the decomposition is more complicated. The query in Figure 7.20 requires customer a be perpetually making a request for at least one of the three keys. I divide this query into three queries each of which requires customer a be requesting one specific key. The query in which customer a repeatedly requests key 1 is shown in Figure 7.22. This is a query for a trace in which customer a is prevented from using the resource, in part, by not being granted key 1 despite repeated requests for it. The toolset reported that no trace matching this query exists. I modify the system so that guards 1 and 2 will not set the wait flag for customer b after customer a is refused the key. When the query was applied to the incorrect system, the toolset found a solution corresponding to a trace in which customer a acquires key 3 but is then repeatedly denied keys 1 and 2, while customer b repeatedly acquires keys 1 and 2 and uses the resource.

The important difference between the queries of Figure 7.20 and Figure 7.22 is that the latter query forces the finite interval to end after a request by customer a for key 1. I can do this without eliminating from consideration any traces I am interested in (i.e., those in which, after some point, customer a requests key 1 and is denied it repeatedly). The presence of this boundary, however, forces guard 1 to

```
(defquery "starve-1" "nofair" (omega-star-less (sequence
  (interval :initial t :open t
            :ends-with '((rend "customer_a;guard_1.request_1a")))
  (interval :perpetual t
            :require '((rend "customer_a;guard_1.request_1a")
                       (rend "customer_b;resource.enter"))
            :forbid '((rend "customer_a;resource.enter")
                      (rend "guard_1;customer_a.grant_1a"))))))
```

Figure 7.22. Simplified Query for Starvation in Version Four

```
(defquery "deadlock" "nofair" (omega-star-less (sequence
  (interval :initial t :final t))))
```

Figure 7.23. Query for Deadlock in Version Four

accept customer a's request in a cycle of its FSA in which customer b has the key; all such cycles must contain customer a's eventual acquisition of the key. In the spurious solution found by the query in Figure 7.20, customer a's request was accepted and declined by an isolated cycle in guard 1's FSA in which customer b does not have the key, but in which `wait_a` is `true`.

While the modifications made to the guard tasks enforce stricter fairness, there is also the risk that they could produce a deadlock in which neither of the customers is using the resource. To verify that this is not possible, I used the query in Figure 7.23 to search for any finite traces. The toolset reported that no traces matching this query exist, proving that a deadlock involving all of the tasks (as any deadlock involving both customers would) is impossible.

In conclusion, the modified toolset was able to give a definitive answer for all the properties I attempted to verify. As Table 7.2 shows, the average analysis time to verify or disprove a property of these systems, starting from the CEDL source and the LISP-like query, was about five minutes. Although larger inequality systems were produced for more complex queries involving multiple intervals, these systems did not prove to be significantly harder to solve. The presence of multiple sequences in a

Table 7.2. Toolset Performance on Coterie Mutual Exclusion

Version	Query	Result	Deriv	Elim	Ineq	ILP	Total	Size
1	mutex	no trace	42	3	28	258	331	234 × 272
1 (i)	mutex	solution	41	2	25	4	72	220 × 254
1	deadlock	no trace	42	3	6	1	52	93 × 102
1	starve	solution	42	3	12	14	71	154 × 150
1	starve-fair	no trace	42	3	12	19	76	172 × 150
1 (i)	starve-fair	solution	54	4	14	3	75	170 × 147
2	mutex	no trace	172	13	179	1140	1504	365 × 414
2 (i)	mutex	solution	179	12	179	8	378	356 × 405
2	queue	solution	172	13	69	2	256	315 × 410
2 (i)	queue	solution	179	12	71	3	265	319 × 419
3	mutex	ILP fails	203	35	91	2483	2812	591 × 661
3	mutex-1	no trace	203	35	42	4	284	381 × 537
3 (i)	mutex	solution	201	34	92	36	363	570 × 638
3	starve	solution	203	35	53	34	325	467 × 511
4	mutex-1	no trace	254	44	62	7	367	552 × 855
4 (i)	mutex-1	solution	260	41	51	5	357	449 × 668
4	starve	spurious	254	44	83	74	455	650 × 742
4	starve-1	no trace	254	44	71	10	379	549 × 598
4 (i)	starve-1	solution	260	40	52	25	377	457 × 484
4	deadlock	no trace	254	44	26	317	641	231 × 334

query, however, did greatly increase the time to solve the resulting inequality system (as shown by the **mutex** queries of versions one, two, and three). This suggests that fixing the order of required event symbols in a trace is not as computationally expensive in the integer programming as specifying a choice between event symbols that must appear, even though the former generally increases the size of the inequality system more.

The performance of the technique as the concurrent systems are scaled up has yet to be explored. Unlike the dining philosophers systems analyzed in [8], the systems analyzed in this section become much more complicated when they are scaled up. Even with three customers that can each request any key, a deadlock is possible (in which each customer has one key) which must be detected and avoided by the system. I chose to analyze these systems rather than the easily scalable dining philosophers systems because these systems are (I believe) more realistic in that they have a few

complex tasks rather than many trivial ones, and they have more interesting safety and liveness properties. Further experiments on systems that can be scaled up and on a variety of different systems are needed to assess the practicality of the techniques, but I believe these experiments demonstrate the feasibility of verifying general safety and liveness properties with integer programming.

7.2.3 Experiments Detecting Critical Races

This section describes a series of experiments with the technique of Section 5.5.1 for detecting a critical race between two tasks.

The two small examples in Figures 5.15 and 5.17 (pages 114 and 117) were coded in CEDL and fed to the toolset along with a query asking whether the apparent race in the resource task between $r1$ and $r2$ was real. In the first example, the toolset found the solution corresponding to the pair of traces described. In the second example, the toolset reported that the generated inequality system had no integral solution. The performance of the toolset on these examples is reported in the first two rows of Table 7.3 (page 189), which shows all the performance data for this section. It lists, for each example and race sought: the result of the experiment, the time (in seconds on a DECstation 5000/125) taken by each of the tools, the total time, and the size of the inequality system generated (number of inequalities $\times$ number of variables).

I also tried the technique on version one of the coterie mutual exclusion example, described in Section 7.2.2 (page 168). I sought to verify that the apparent race in the resource between customers a and b entering is not real (i.e., the mutual exclusion enforced by the guards should never allow more than one customer to be ready to use the resource at any given time). When fed the query in Figure 7.24, which matches a pair of prefixes showing a race between customers a and b at the `ENTER` entry of the resource task, the toolset reported that the necessary conditions for this race to be real are unsatisfiable, proving that the race is only apparent. I applied the same query to the incorrect version of this system, described in Section 7.2.2, that does

```
(defquery "race" "nofair"
  (critical-race :task "resource"
     :rend1 "customer_a;resource.enter"
     :rend2 "customer_b;resource.enter"))
```

Figure 7.24. Query for Race in Coterie Mutual Exclusion System

not enforce mutual exclusion. For this incorrect system, the toolset found a solution corresponding to a pair of prefixes showing a race.

The last example on which I tried the technique is a version of the gas station example, first introduced in [38] and analyzed in [8]. The CEDL for this system is shown in Figures 7.25 and 7.26. The system consists of an operator, a pump, and two customers. Each customer signals the operator to prepay, then signals the pump to start and finish pumping. The pump signals the operator when a customer finishes pumping, and the operator then signals the customer with his/her change. A critical race exists in this system at the START_PUMPING entry of the pump task between two customers who have both prepaid. If the customer who prepaid second uses the pump first, he/she will receive the change of the customer who prepaid first. In another version of this system analyzed in [8], this race can lead to deadlock. In this version, however, the customers may simply receive the wrong change.

I may assume that the intent of the designer was to allow the customers to prepay in any order (since they could arrive at the station in any order), and then have the customers pump gas in the order that they paid. I first attempted to verify that the selection of the next customer to prepay is nondeterministic. When fed a query for a race between the customers at the PREPAY entry, the toolset found a solution corresponding to a pair of prefixes showing a race. I then attempted to verify that the selection of the next customer to pump gas is deterministic. However, when fed a query for a race between the customers at the START_PUMPING entry of the pump task, the toolset again found a solution corresponding to a pair of prefixes showing a race. This exposes the critical race.

```
package COMMON is
  type C_NAME is (c1,c2);  -- names for two customers
  type COUNTER is (zero,one,two,three);
end COMMON;

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 waiting
            CURRENT := CUSTOMER_ID;  -- mark this one as current
            PUMP.ACTIVATE;           -- and activate the pump
           else
            WAITING := CUSTOMER_ID;   -- otherwise, this one next
          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 = c1 then
          CUSTOMER_1.CHANGE1;
         else
          CUSTOMER_2.CHANGE2;
        end if;
        CUSTOMERS := COUNTER'pred(CUSTOMERS);
        if CUSTOMERS > zero then   -- if another customer is
          CURRENT := WAITING;      -- waiting, that one now current
        end if;
      end select;
    end loop;
  end OPERATOR;
```

Figure 7.25. CEDL for Two Customer Gas Station (Part 1)

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

use COMMON;
task body CUSTOMER_1 is
begin
  loop
    OPERATOR.PREPAY(c1);
    PUMP.START_PUMPING;
    PUMP.FINISH_PUMPING;
    accept CHANGE1;
  end loop;
end CUSTOMER_1;

use COMMON;
task body CUSTOMER_2 is
begin
  loop
    OPERATOR.PREPAY(c2);
    PUMP.START_PUMPING;
    PUMP.FINISH_PUMPING;
    accept CHANGE2;
  end loop;
end CUSTOMER_2;
```

Figure 7.26. CEDL for Two Customer Gas Station (Part 2)

Table 7.3. Toolset Performance on Race Detection

Example	Race	Result	Deriv	Elim	Ineq	ILP	Total	Size
Alt(i)	use	solution	24	1	2	1	28	37 × 33
Alt	use	no race	26	1	3	1	31	39 × 34
Cot1	use	no race	42	3	16	1	62	131 × 120
Cot1(i)	use	solution	41	2	13	1	57	123 × 114
Gas	prepay	solution	46	16	35	2	99	207 × 216
Gas	pump	solution	46	16	38	4	104	233 × 301

These experiments demonstrate the feasibility of detecting and proving the absence of critical races with the technique of Section 5.5.1. As with the other techniques, more experiments on a variety of types and sizes of systems are necessary to establish the practicality of the technique.

7.2.4 Experiments Deriving Uniprocessor Bounds

This section describes a series of experiments with the technique for deriving upper and lower bounds on the time that can elapse between events in a concurrent real-time system run in a uniprocessor setting. As I indicated in the description of this technique in Section 6.1, the inequalities for the technique can be generated using the method of Section 5.1.1 and I used queries that look like those in Section 7.2.2. Queries for obtaining uniprocessor bounds differ from those normally used for concurrency analysis in that I specify the objective function explicitly for each interval rather than allowing the weight of each ILP variable to default to one.

First consider the gas station example given in Figures 7.25 and 7.26 (pages 188 and 189). Suppose that the duration of each event is one time unit and that an upper bound on the time that can elapse between customer 1 prepaying and receiving change is required. The query for this prefix is shown in Figure 7.27. Since I seek a prefix of a trace, I can forbid the hang symbols, as noted in Section 7.2.2. The durations of the event symbols for the objective function are specified using the `:costs` parameter. This query specifies that all transition variables in the first interval have weight zero in the objective function and that all transition variables in the second interval have weight zero, except `call` symbols, which have weight -1. The ILP package is set to minimize, so using negative weights for the events causes a timed interval of maximum duration to be selected. Finally, note that I specify the start event in the timed interval rather than in the initial interval. This is only for the benefit of the marking algorithm; the start event is still in the initial interval and the connection variables are on the same states as they would be if the initial interval were specified as ending

```
(defquery "max" "nofair" (pattern (sequence
 (interval :initial :t :open t :forbid '((prefix "hang"))
           :costs '(costs (:base 0)))
  (interval :forbid '((prefix "hang"))
            :starts-after '((rend "customer_1;operator.prepay;c1"))
            :ends-with '((rend "operator;customer_1.change1"))
            :costs '(costs (:base 0)
                           ((prefix "call") -1))))))
```

Figure 7.27. Query for Longest Interval Between Customer One Prepaying and Receiving Change

with the start event. Since an invocation of the marking algorithm on the FSAs of an interval uses only start and halt events specified in that interval, specifying both the start and halt events in the timed interval makes the marking algorithm more effective on that interval, where the removal of cycles is most important.

When fed this query, the toolset found a solution corresponding to a prefix of a trace in which customer 2 prepays, then customer 1 prepays (the timed interval begins), customer 2 pumps gas and receives change, customer 2 prepays again and pumps gas, and customer 1 pumps gas and receives change (the timed interval ends). The segment found for the timed interval has 13 communications and is the longest segment of a trace between the start and halt events, thus the upper bound of 13 is sharp. The performance of the toolset is given, along with all the performance data in this section, in Table 7.4 (page 198). It lists, for each example and query: the result of the experiment, the time (in seconds on a DECstation 5000/125) taken by each of the tools, the total time, and the size of the inequality system generated (number of inequalities $\times$ number of variables).

Using the same durations, I also used the tool to derive a lower bound on the time that can elapse between two consecutive activations of the pump. The query for this bound is shown in Figure 7.28. Note that I use positive weights to minimize the duration of the timed interval. The toolset produced a sharp bound of 5 by finding a solution that corresponds to a prefix of a trace in which customer 2 prepays, the

```
(defquery "min" "nofair" (pattern (sequence
  (interval :initial :t :open t :forbid '((prefix "hang"))
            :costs '(costs (:base 0)))
  (interval :forbid '((prefix "hang"))
            :starts-after '((rend "operator;pump.activate"))
            :ends-with '((rend "operator;pump.activate"))
            :costs '(costs (:base 0)
                           ((prefix "call") 1))))))
```

Figure 7.28. Query for Shortest Interval Between Successive Pump Activations

pump is activated (the timed interval begins), customer 1 prepays and pumps gas, the pump reports the charge to the operator, and the pump is activated again (the timed interval ends). This prefix has the same duration (in the timed interval) as simpler ones in which one customer pumps twice while the other does nothing.

The second example I analyzed was a system with a pool of four resources, a controller for this pool, and three customers that communicate among themselves in addition to using the resources. Customer 1 repeatedly chooses nondeterministically between two courses of action. The first course involves communicating with customer 2, obtaining access to the resources from the controller, using resource 1, and then relinquishing access. The second involves communicating with customer 3, obtaining access to the resources, using resource 2, and then relinquishing access. Customer 2 alternates between using resources 1 and 3, gaining and relinquishing access to the resource pool for each use, and communicating with customer 1 before using resource 1. Customer 3 is like customer 2, except that it alternates between using resources 2 and 4. The resource controller allows no more than two customers to have simultaneous access to the resource pool. It also enforces fairness by the simple strategy of requiring the customers to release access to the resource pool in the same order as they acquired it. The CEDL code for this system is given in Figures 7.29 and 7.30.

In this system, I assigned all events zero duration except for the use of the resources, which I assigned an integer duration equal to the index of the resource

```
package COMMON is
  type C_NAME is (c1,c2,c3,empty);  -- names for three customers, plus empty
end COMMON;

use COMMON;
task CONTROL is
  entry ENTER(CUSTOMER_ID: in C_NAME);
  entry EXIT1;
  entry EXIT2;
  entry EXIT3;
end CONTROL;

task body CONTROL is
CURRENT, WAITING: C_NAME := empty;
begin
  loop
    select
      when (waiting = empty) =>
        accept ENTER(CUSTOMER_ID: in C_NAME) do
          if (current = empty) then
            current := CUSTOMER_ID;
          else
            waiting := CUSTOMER_ID;
          end if;
        end ENTER;
    or
      when (current = c1) =>
        accept EXIT1;
        current := waiting; waiting := empty;
    or
      when (current = c2) =>
        accept EXIT2;
        current := waiting; waiting := empty;
    or
      when (current = c3) =>
        accept EXIT3;
        current := waiting; waiting := empty;
    end select;
  end loop;
end CONTROL;
```

Figure 7.29. CEDL for Resource Pool Example (Part 1)

```
use COMMON;
task CUSTOMER_1;

task body CUSTOMER_1 is
Environment : BOOLEAN;
begin
  loop
    Environment := ...;
    if Environment then
      CUSTOMER_2.A;
      CONTROL.ENTER(c1);
      R1.USE1;
      CONTROL.EXIT1;
    else
      CUSTOMER_3.B;
      CONTROL.ENTER(c1);
      R2.USE2;
      CONTROL.EXIT1;
    end if;
  end loop;
end CUSTOMER_1;
```

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

task body CUSTOMER_2 is
begin
  loop
    CONTROL.ENTER(c2);
    accept A;
    R1.USE1;
    CONTROL.EXIT2;
    CONTROL.ENTER(c2);
    R3.USE3;
    CONTROL.EXIT2;
  end loop;
end CUSTOMER_2;
```

```
use COMMON;
task CUSTOMER_3 is
  entry B;
end CUSTOMER_3;

task body CUSTOMER_3 is
begin
  loop
    CONTROL.ENTER(c3);
    accept B;
    R2.USE2;
    CONTROL.EXIT3;
    CONTROL.ENTER(c3);
    R4.USE4;
    CONTROL.EXIT3;
  end loop;
end CUSTOMER_3;
```

```
task R1 is
  entry USE1;
end R1;

task body R1 is
begin
  loop
    accept USE1;
  end loop;
end R1;
```

```
task R2 is
  entry USE2;
end R2;

task body R2 is
begin
  loop
    accept USE2;
  end loop;
end R2;
```

```
task R3 is
  entry USE3;
end R3;

task body R3 is
begin
  loop
    accept USE3;
  end loop;
end R3;
```

```
task R4 is
  entry USE4;
end R4;

task body R4 is
begin
  loop
    accept USE4;
  end loop;
end R4;
```

Figure 7.30. CEDL for Resource Pool Example (Part 2)

```
(defquery "max-1" "nofair" (pattern (sequence
  (interval :initial :t :open t :forbid '((prefix "hang"))
            :costs '(costs (:base 0)))
  (interval :forbid '((prefix "hang"))
            :starts-after '((rend "customer_1;control.enter;c1"))
            :ends-with '((rend "customer_1;control.exit1"))
            :costs '(costs (:base 0)
                           ("call(customer_1;r1.use1)" -1)
                           ("call(customer_2;r1.use1)" -1)
                           ("call(customer_1;r2.use2)" -2)
                           ("call(customer_3;r2.use2)" -2)
                           ("call(customer_2;r3.use3)" -3)
                           ("call(customer_3;r4.use4)" -4))))))
```

Figure 7.31. Query for Longest Interval Between Customer One Entering and Exiting Resource Pool

(i.e., using resource i takes i time units). I sought an upper bound on the time that can elapse between the first customer entering and exiting the resource pool. The query for this prefix is shown in Figure 7.31. When fed this query, the toolset produced a sharp bound of 9 by finding a solution corresponding to the prefix of a trace represented in Figure 7.32 by the actions of each customer.

The prefix represented in Figure 7.32 has an initial segment that is longer than necessary to achieve the timed segment of greatest duration (customer 3 need not complete a cycle before the timed interval begins). For all upper/lower bound queries, it would be desireable to minimize the values of the transition variables in the initial interval, since this makes it easier to check whether the solution corresponds to a segment of a trace. This could be accomplished using a standard optimization technique as follows: find a solution optimizing the duration of the timed interval, add a constraint to the inequality system setting the expression for the duration of the timed interval to this optimal value, and then find a solution of this augmented system that minimizes the number of events in the initial interval. Although straightforward, this enhancement has not yet been implemented.

For comparison, I tried to derive the same bound without using an initial interval, as described in Section 6.1.3. I used a query called `max-ni`, which looks exactly like the

Customer 1	Customer 2	Customer 3
	customer_2;control.enter	
customer_1;customer_2.a	customer_1;customer_2.a	
	customer_2;r1.use1	
	customer_2;control.exit2	
customer_1;control.enter		
customer_1;r1.use1		
customer_1;control.exit1		
		customer_3;control.enter
customer_1;customer_3.b		customer_1;customer_3.b
		customer_3;r2.use2
		customer_3;control.exit3
		customer_3;control.enter
		customer_3;r4.use4
		customer_3;control.exit3
customer_1;control.enter		
customer_1;r2.use2		
customer_1;control.exit1		
		customer_3;control.enter
customer_1;customer_3.b		customer_1;customer_3.b
		customer_3;r2.use2
		customer_3;control.exit3
	customer_2;control.enter	
customer_1;control.enter		
(timed interval begins)	(timed interval begins)	(timed interval begins)
customer_1;r2.use2		
	customer_2;r3.use3	
	customer_2;control.exit2	
		customer_3;control.enter
		customer_3;r4.use4
customer_1;control.exit1		

Figure 7.32. Prefix of Trace of Resource Pool Example Produced by Query max-2

query `max-1` in Figure 7.31 except that the first interval has been removed. When fed this query, the toolset reported an upper bound of 11 corresponding to an unreachable segment in which all three customers are in the resource pool at the start of the timed interval, allowing customer 3 to use resource 2 one extra time (this segment can be obtained from the prefix of the trace in Figure 7.32 by including customer 3's use of resource 2 in the timed interval and then removing all events not included in the timed interval). As reported in Table 7.4, the inequality system for this query had fewer than half as many inequalities as the one with the initial interval and took half as much time to solve.

Finally, I tried to derive the same bound assuming resource 1 takes five time units rather than one. This query, `max-2`, looks exactly like query `max-1` in Figure 7.31 except that the weights for the `call` symbols for resource 1 are -5. When fed this query, the toolset produced a sharp bound of 14 corresponding to a prefix of a trace in which resource 1 is used by customers 1 and 2 and resource 4 is used by customer 3 within the timed interval.

The last concurrent system to which I applied the technique was the second version of the coterie mutual exclusion system from Section 7.2.2 (page 173). For this system, I sought an upper bound on the time that can elapse between customer a requesting key 2 (its second key) and starting to use the resource, assuming all communications take unit time. I formulated this query as shown in Figure 7.33 and fed it to the toolset. Unfortunately, the solution found by the toolset yields no bound due to unbounded spurious cyclic flows in which customer c repeatedly uses the resource. (In practice, I always set a large upper bound on the integer programming variables, but if any of the variables are set at or near this bound in a solution, I interpret the solution as unbounded).

The marking algorithm is unable to remove cycles in FSAs for which it has no start or halt events. In this case, the cycles through which the unbounded flow is passing are in customer c, guards 1 and 3, and the resource. Of these, the cycle

```
(defquery "max-1" "nofair" (pattern (sequence
  (interval :initial :t :open t :forbid '((prefix "hang"))
            :costs '(costs (:base 0)))
  (interval :forbid '((prefix "hang"))
            :starts-after '((rend "customer_a;guard_2.request_2a"))
            :ends-with '((rend "customer_a;resource.enter"))
            :costs '(costs (:base 0)
                           ((prefix "call") -1))))))
```

Figure 7.33. Query for Longest Interval Between Customer a Requesting its Second Key and Using the Resource

Table 7.4. Toolset Performance on Uniprocessor Bounds

Example	Query	Result	Deriv	Elim	Ineq	ILP	Total	Size
Gas	max	sharp	46	17	32	2	97	129 × 171
Gas	min	sharp	46	17	34	2	99	187 × 273
Pool	max-1	sharp	73	12	21	2	108	166 × 230
Pool	max-ni	not sharp	73	12	11	2	98	66 × 117
Pool	max-2	sharp	73	12	21	2	108	166 × 230
Cot2	max-1	no bound	172	12	37	3	224	248 × 334
Cot2	max-2	sharp	172	12	36	3	223	218 × 284

through guard 1 is the one that cannot legitimately occur since customer a must just have acquired key 1 before the timed interval begins (therefore guard 1 cannot grant key 1 to customer 3). I can help the marking algorithm remove this cycle by specifying `resume(guard_1;customer_a.grant_1a)` as an additional start event of the timed interval. I constructed the query `max-2` from query `max-1` in Figure 7.33 by adding this additional start event. When fed to the toolset, this query produced a sharp bound of 11 corresponding to a prefix of a trace given by the actions of the customers in Figure 7.34.

These experiments demonstrate the feasibility of deriving bounds on concurrent real-time systems run in a uniprocessor setting using the technique of Section 6.1. When seeking upper bounds, cycles in the FSAs can seriously degrade the bound obtained, but, in my experience, the marking algorithm of Section 7.1.2 often solves this problem by removing unreachable cycles before inequality generation. Finally,

```
                    Customer a:
                      customer_a;guard_1.request_1a
                      guard_1;customer_a.grant_1a
                      customer_a;guard_2.request_2a
                      (timed interval begins)
                      guard_2;customer_a.grant_2a
                      customer_a;resource.enter

Customer b:                          Customer c:
  customer_b;guard_2.request_2b        customer_c;guard_1.request_1c
  guard_2;customer_b.grant_2b          guard_1;customer_c.grant_1c
  (timed interval begins)              customer_c;guard_3.request_3c
  customer_b;guard_3.request_3b        guard_3;customer_c.grant_3c
  guard_3;customer_b.grant_3b          customer_c;resource.enter
  customer_b;resource.enter            customer_c;resource.leave
  customer_b;resource.leave            customer_c;guard_1.free_1c
  customer_b;guard_2.free_2b           (timed interval begins)
  customer_b;guard_3.free_3b           customer_c;guard_3.free_3c
  customer_b;guard_2.request_2b        customer_c;guard_1.request_1c
```

Figure 7.34. Prefix of Trace of Coterie Mutual Exclusion Example Produced by Query `max-1`

note that the technique's assumption of worst case scheduling makes upper bounds difficult to obtain except in tightly coupled systems. For example, in the coterie mutual exclusion system, there are few pairs of events that do have a finite upper bound on the time that can elapse between them. Assumptions of fairness that promise progress only in the limit do not help in the derivation of these bounds.

7.2.5 Experiments Deriving Multiprocessor Bounds

This section describes a series of experiments with the technique of Section 6.2 for deriving an upper bound on the time that can elapse between two events in a concurrent real-time program run in a maximally-parallel multiprocessor setting. The performance data for the toolset are aggregated in Table 7.5 (page 205). It lists, for each example and query: the result of the experiment, the time (in seconds on a DECstation 5000/125) taken by each of the tools, the total time, and the size of the inequality system generated (number of inequalities $\times$ number of variables).

```
(defquery "max-1" "nofair"
  (parallel-execution (sequence
     (interval :initial t :final t
               :forbid '((prefix "hang"))
               :costs '(costs (:base 0)
                              ((rend "customer_1;resource.use_it") -1)
                              ((rend "customer_2;resource.use_it") -1)
                              ("use(customer_1;choice;false)" -1)
                              ("use(customer_2;choice;false)" -1))))))
```

Figure 7.35. Query for Longest Execution in Maximally-Parallel Setting

The resource contention example presented in Section 6.2.2 (page 136) was coded in CEDL and fed to the toolset along with queries like those in Figure 7.35. The use symbols represent the action a customer takes if it chooses not to use the resource. I derived bounds for the three cases described in Section 6.2.2 by adjusting the durations of the event symbols specified in the query. In query max-1, as shown, all events take unit time; in query max-2, customer 1's use of the resource takes 10 time units instead; in query max-3, all events take unit time except customer 1's action when not using the resource, which takes 10 time units. When these three queries were fed to the toolset, a sharp upper bound is obtained in each case, as described in Section 6.2.2, and the wait path specified by the solution is a critical path.

The second experiment involves a remote server system, shown in Figure 7.36. The system consists of a handler, a local server, a remote server, a relay, and a resource. I model a single invocation of the handler after receiving a request to prepare the resource for use (e.g., open a file). The resource could either be at the same site or at another site. The handler determines the current location of the resource (this is modeled as a nondeterministic choice by the handler) and either starts a local server or starts a remote server and a relay. The local server, if activated, does a local computation, activates the resource, and reports to the handler that it is done. The remote server, if activated, does a local computation, activates the resource, and

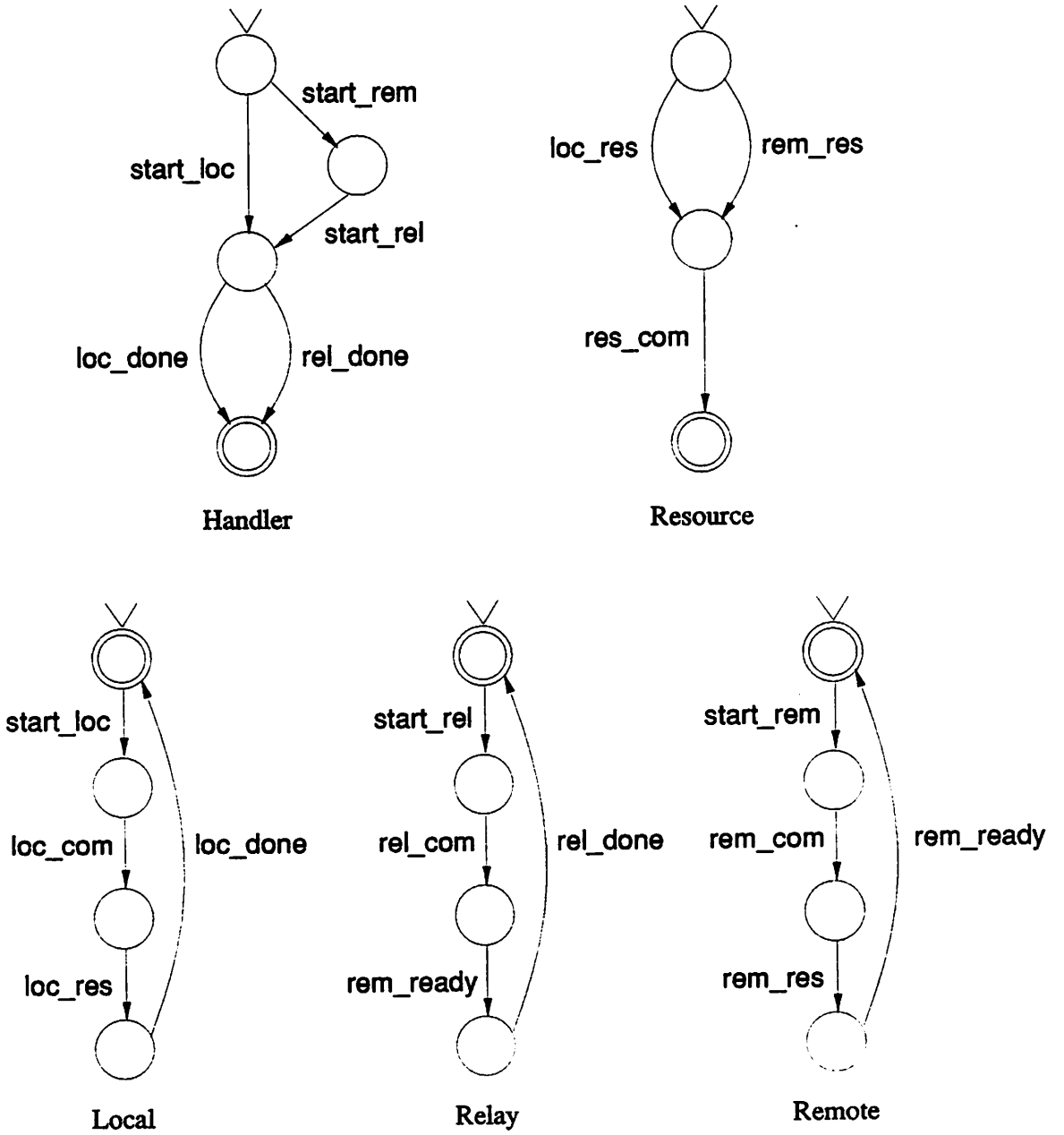


Figure 7.36. Remote Server Example

reports to the relay that it is done. The relay, if activated, does a local computation, waits for the remote server to finish, and then reports to the handler that the remote connection to the resource (i.e., through the relay) is complete. The resource, when activated, simply does a local computation.

I sought an upper bound on the execution time of this system in a maximally-parallel multiprocessor setting assuming that all communications take 2 time units and the internal computations of the tasks take 10 time units, except for that of the local server, which takes 20 time units. One feature of this system that makes it interesting is that the execution having the longest execution time on a uniprocessor is not the same as the execution having the longest time on a multiprocessor. Setting up the use of the resource at a remote site takes more total computation, but on a multiprocessor much of this computation proceeds in parallel.

When the toolset was fed the CEDL for this system and a query for a bound on the time of longest execution on a uniprocessor, it reported a bound of 50, which is sharp, corresponding to an execution in which the resource is at a remote site. (Note that on a uniprocessor, a communication takes 2 time units in each task.) When I fed the toolset a query for a bound on the longest execution on a multiprocessor, the toolset reported a bound of 40 that is not sharp due to a spurious cyclic flow in the potential wait graph. The wait path found by this solution starts in the handler, passes from the handler to the remote server through the `start_rem` communication, and then passes into the resource through the `rem_res` communication. An additional cyclic flow of one passes through the cycle in the relay FSA.

To prevent this cyclic flow from degrading the bound, I recoded the example, unrolling each loop in a task the maximum number of times it could be executed (which, in this case, is one). This removes all the cycles from the FSAs and the potential wait graph. When the multiprocessor bound query was applied to this version, the toolset reported a sharp upper bound of 34 corresponding to an execution in which the resource is local. The wait path selected by this solution, which is a

critical path, starts in the local server and passes into the resource at the `loc_res` communication.

A third example I analyzed was the reactor monitoring system in Figure 7.37. The system is composed of four control tasks and six resource tasks. The four control tasks are shown in Figure 7.37; the six resource tasks are not shown but are modeled just as the resource in the resource contention example of Figure 6.4 (i.e., without loops, using a bound on the number of times the resource can be used). The resources are: a thermometer, a panel display, a log, a pressure sensor, a valve (accepting either an open or a close command), and an alarm bell. The event symbols representing use of the resources (i.e., communication with tasks not shown) are in boldface. Several parentheticals have been added at branches in the FSAs; these are simply comments indicating what the corresponding branch means in terms of the reactor (e.g., pressure acceptable). The control tasks read the temperature and pressure of the reactor and record the data on the log, display information on the control panel, etc.

I sought a bound on the execution time given the following event durations: use of a resource takes one time unit, except for using the log, which takes 10, and opening the valve, which takes 5. All other events take no time. When fed a query (`max-1`) for an upper bound in the multiprocessor setting, the toolset produced a bound of 23, which is not sharp. The wait path corresponding to the solution starts at the top of task `hot`, crosses over to task `main` through the use of the panel resource, and then enters the log resource. This is not a feasible wait path because of the communication event matchings in the panel resource. The use of the panel by task `main` must occur before the use of the panel by the task `temp` because of the `start` communication (in fact, there can be no contention over this resource even though there are two users). Therefore, the use of the panel by task `hot` cannot match the first use of the panel in its resource task, as in the wait path found. The least upper bound is 22, given by a critical path through an execution where the temperature is too high but the pressure is acceptable.

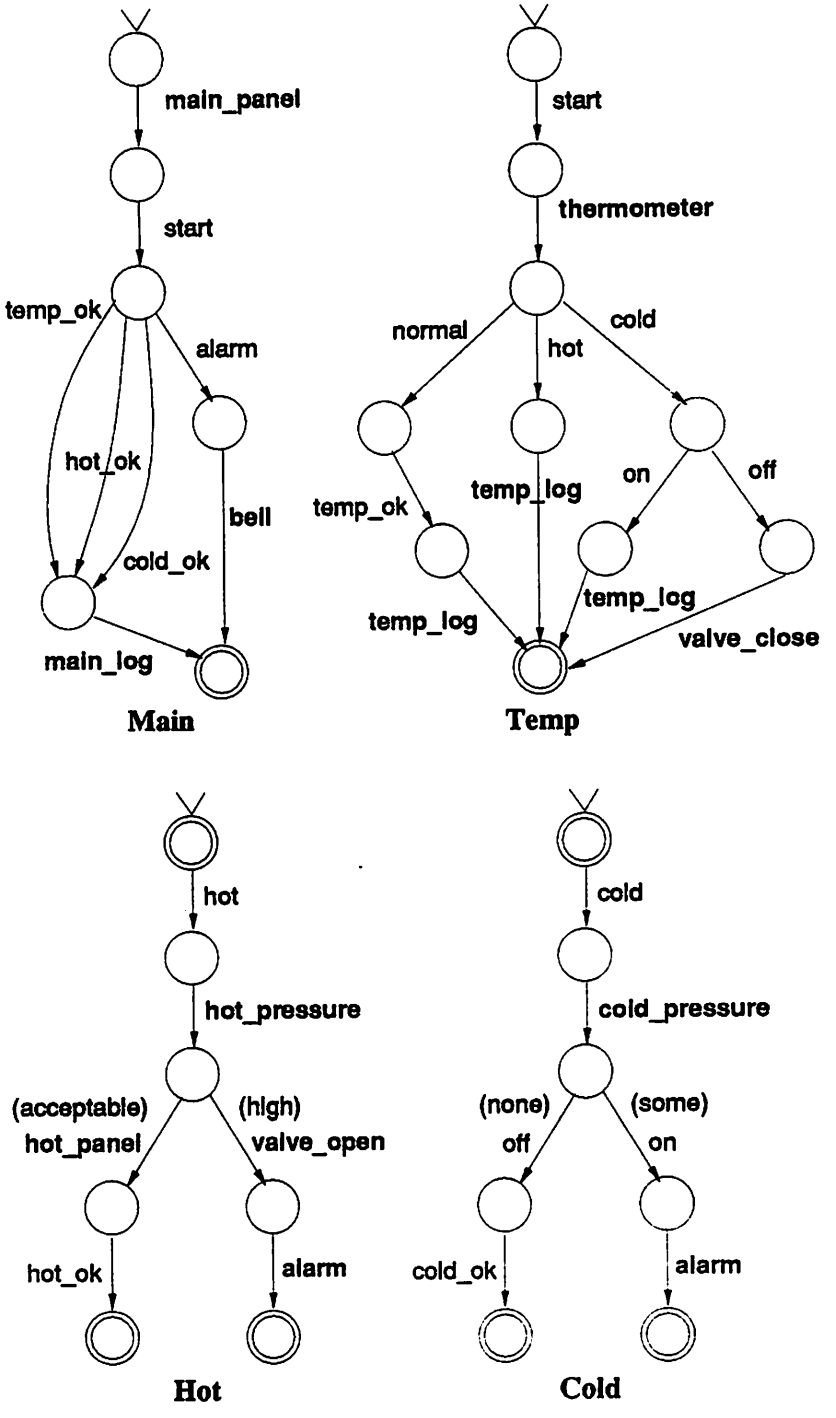


Figure 7.37. Reactor Monitor Example

Table 7.5. Toolset Performance on Multiprocessor Bounds

Example	Query	Result	Deriv	Elim	Ineq	ILP	Total	Size
Con	max-1	sharp	99	1	5	1	105	80 × 66
Con	max-2	sharp	99	1	4	1	105	80 × 66
Con	max-3	sharp	99	1	4	1	105	80 × 66
Rem	uni	sharp	47	3	4	1	55	40 × 59
Rem	multi	not sharp	47	3	10	2	62	215 × 162
Rem(u)	multi	sharp	46	3	9	2	60	178 × 134
Mon	max-1	not sharp	164	5	28	7	204	487 × 365
Mon	max-2	not sharp	164	5	27	103	299	487 × 365

Here I could use the third technique of Section 6.2.4 to eliminate this spurious solution. If I split the timed interval at the `start` communication by generating two timed intervals, the first of which ends with `start`, then this infeasible matching would not be possible since only the main task's use of the panel could occur in the first interval. Also, the inequality system generated for the analysis would not be significantly larger in this case, since the marking algorithm would, for each interval, remove almost all of the transitions that could not occur in that interval. In this case, the use of two timed intervals would be like removing the cross arcs representing the infeasible matching.

I also sought an upper bound given that using any resource takes unit time, except closing the valve, which takes 20 time units. When fed this query (`max-2`), the toolset produced a sharp upper bound of 23 corresponding to an execution in which the temperature is too low and the pressure is acceptable. The wait path given by the solution is a critical path, starting in task `main`, crossing over to task `temp` through the `start` communication, then crossing to task `cold` through the `cold` communication, crossing back to task `temp` through the `off` communication, and passing through the `valve_close` communication.

The final experiment I describe is with the second method for tightening the bounds presented in Section 6.2.4. This method sought to iterate through the solutions of a restricted type of integer system until a solution is reached that yields a

bound that is known to be sharp. I have implemented a small part of this method. When instructed, the toolset can automatically generate the inequality excluding the previous solution. I tried this method on the first query of the reactor monitor example, which yielded a spurious wait path. The second solution found by the integer programming package, although different from the first, had essentially the same wait path due to the great amount of symmetry in the potential wait graph. The third and fourth solutions also yielded wait paths that were essentially equivalent to the first. Unfortunately, there are a huge number of different but essentially equivalent wait paths through each execution in that system. Even in the resource contention example of Figure 6.4, the smallest example I analyzed, there are a total of eight possible critical paths for the two executions in which both customers use the resource. Therefore, I believe that any successful exploitation of this method must have a way to account for such symmetries, perhaps by generating an inequality (or inequalities) that excludes both a solution and all related solutions. Inequalities involving the total numbers of event symbols rather than transitions are a prime candidate, but these quantities need not be boolean, and so I encounter the problem mentioned in Section 6.2.4.

The experiments described above demonstrate the feasibility of deriving bounds for some concurrent real-time systems run in a maximally-parallel multiprocessor setting using the technique of Section 6.2. While the experiments showed that sharp upper bounds can be obtained for some systems, they also revealed that cycles in the potential wait graph can degrade the bound. Unlike the cycles encountered in the uniprocessor technique, these cycles usually cannot be removed using a graph marking algorithm like the one described in Section 7.1.2. I believe that devising a method to deal with cyclic flows in the potential wait graph will be essential in deriving quality bounds for concurrent systems with cycles in their FSAs.

7.3 Summary

In this chapter, I described the implementation of a prototype tool that automates many of the analysis techniques of the preceding chapters, and I described a series of experiments carried out with this tool on a number of sample concurrent systems. While the experiments described in this chapter demonstrate the feasibility of the techniques tested, they are not sufficient to determine their practical significance. A much more extensive empirical evaluation of these techniques, involving the their application to a variety of types and sizes of systems, will be the subject of future work.

CHAPTER 8

CONCLUSION

Almost all automated verification of concurrent systems for properties other than deadlock involves some form of model checking. Unfortunately, in most realistically-sized systems, the state explosion problem makes the size of the models too large for the application of standard model checking procedures. Although some techniques have shown a prowess for certain types of systems (e.g., symbolic model checking [15] can verify addition circuits with huge state spaces), none have demonstrated practical applicability to a wide range of systems. My techniques allow the automated verification of certain properties of a system without generating the state space of the system. Unlike model checking procedures, my techniques cannot always determine whether or not a property holds, but because they do not enumerate the system's states, they may be applicable to much larger systems. Further experimentation is needed to determine how scalable and widely-applicable the techniques are, but given the demonstrated scalability and wide-applicability of the original constrained expression technique on which they are based, I believe the techniques will outperform their model checking rivals on some large systems.

Specifically, the techniques I have presented extend the constrained expression analysis technique for concurrent and real-time systems to verify many important types of properties it was previously unable to address, including liveness properties, properties involving the order of events, and critical races. In addition, I have shown how to greatly improve the efficiency of the technique when analyzing systems with a large number of identical tasks. Finally, I have completely automated the derivation of bounds for the time between events in a concurrent real-time program run on a

uniprocessor, and I have extended this work to the maximally parallel multiprocessor setting.

The importance of these extensions lies in the added capability to verify many properties of interest to systems designers using a technique that does not require the enumeration of the state space of a concurrent system and may therefore be able to serve as the basis for practical analysis tools. The original constrained expressions analysis technique has been successfully applied to a variety of types and sizes of systems [8], mostly to detect or prove the absence of deadlock. Although the extensions presented here have not been as extensively tested, they have at least shown the feasibility of verifying more complex properties using this promising technique.

Despite this success, however, some problems still remain. As we saw in Section 7.2.1, the technique used to represent identical tasks is very susceptible to numerical instability. In fact, examples such as the dining philosophers, whose communication pattern causes the technique to produce a banded matrix, have also given us stability problems [8]. In practice, there must be a way to tell whether numerical instability has made the output of the integer programming package unreliable.

My experiments with ω-star-less queries in Section 7.2.2 raise questions about the way in which the property to be verified is specified. Clearly the current query language is user-hostile. It allows the analyst to make subtle errors in the query that can result in unsatisfiable necessary conditions, which can in turn lead to erroneous conclusions about the possible behavior of the concurrent program. There are at least two possible solutions to this problem. I could design a query language closer in syntax to regular expressions that hides the division of the trace into intervals and has a simpler semantics than the ω-star-less queries of Section 7.2.2 (e.g., a semantics based on projection only, as in Section 5.1.1). This might reduce the efficiency of the analysis in some cases. Alternatively, I could use temporal logic as my specification language and then translate the formula into an ω-star-less query. This translation

can be done automatically, but may not yield small ω-star-less expressions and so may not be practical.

Other questions involve the decomposition of queries needed in a few cases to successfully complete the analysis. Sometimes this involves simply breaking up disjunctions, but other times a more complex case analysis is required. Can this part of the analysis be formalized? Can it be automated?

Finally, all of these techniques inherit problems from the original technique. The conditions used are necessary but not sufficient, therefore spurious solutions to the inequality systems can make the analysis inconclusive. Also, the use of the constraint eliminator to deal with dataflow is only practical for tasks that use relatively few variables having small ranges to determine what communications in which to engage. This may not be the case for some concurrent programs, especially complex distributed algorithms. Lastly, integer linear programming is NP-hard, although the inequality systems produced by the technique, being largely network flow, seem to have a benign structure that makes them generally easy to solve. Nevertheless, the extensions to the technique, especially the multiprocessor real-time extension, produce inequality systems having a smaller proportion of pure network flow equations that may thus prove more difficult to solve.

All of these problems are areas for future research. In addition, I plan to implement the technique of Section 5.4 for answering queries expressed with an FSA or a Büchi automaton. The bounds derived by the uniprocessor real-time technique could be sharpened by taking into account information on the scheduler, and the multiprocessor technique should be extended to allow multiple tasks on each processor. Also, I intend to investigate methods for proving more complex timing properties of real-time systems, like those that can be expressed in real-time logics.

It is my hope that this work will eventually improve the quality of real-world software systems by bringing rigorous formal methods to bear on these systems through the use of practical analysis tools.

APPENDIX A

ABBREVIATIONS

accept(c)	transitions representing accept on channel c
alphabet(M)	alphabet of FSA M
arcs(G)	arcs of graph G
call(c)	transitions representing call on channel c
caller(c)	FSA that is the caller of channel c
dec(v)	transitions representing decrement of counter v
final(M)	accepting states of FSA M
from(i)	state/node where transition/arc i originates
fsa(i)	FSA containing state/transition i
hang_a(c)	transitions in acceptor representing blockage on channel c
hang_c(c)	transitions in caller representing blockage on channel c
in(j)	transitions into state j
inc(v)	transitions representing increment of counter v
label(k)	event symbol labeling transition/arc k
nodes(G)	nodes of graph G
out(j)	transitions out of state j
over(v)	transitions representing overflow of counter v
occur(e)	transitions labeled with event e in any one FSA containing e
start(j)	true if state j is a start state of an automata, else false
states(M)	states of FSA M
symbols(c)	event symbols used to represent communication on channel c
to(i)	state/node where transition/arc i goes
trans(M)	transitions in FSA M
under(v)	transitions representing underflow of counter v

APPENDIX B

QUERY LANGUAGE SYNTAX

```
(defquery <name of query> <fair> <query>)

<fair> = "fair" | "nofair"

<query> = (omega-star-less <sequence> <sequence> . . .)
        | (parallel-execution <sequence>)
        | (critical-race :task <task name>
                         :rend1 <rendezvous>
                         :rend2 <rendezvous>)

<sequence> = (sequence <interval> <interval> . . .)

<interval> = (interval [:initial <boolean>]
                       [:final <boolean>]
                       [:perpetual <boolean>]
                       [:open <boolean>]
                       [:starts-after <symbol list>]
                       [:ends-with <symbol list>]
                       [:require <symbol list>]
                       [:forbid <symbol list>]
                       [:restrict <list of restriction lists>]
                       [:costs <cost spec>])

<symbol list> = (<symbol spec> <symbol spec> . . .)

<symbol spec> = <symbol>
              | (rend <rendezvous>)
              | (prefix <string>)

<rendezvous> = <caller> ; <acceptor> . <entry>

<list of restriction lists> = (<restriction list>
                               <restriction list> . . .)

<restriction list> = (<relation> (total <symbol> <symbol> . . .)
                                 <number>)
```

```
<cost spec> = (costs ([(:base <number>)]
                      (<symbol spec> <number>)
                      (<symbol spec> <number>) ...))
```

BIBLIOGRAPHY

[1] W. Richards Adrion, Martha A. Branstad, and John C. Cherniavsky. Validation, verification, and testing of computer software. *ACM Computing Surveys*, 14(2):159–192, June 1982.

[2] S. Aggarwal, C. Courcoubetis, and P. Wolper. Adding liveness properties to coupled finite-state machines. *ACM Transactions on Programming Languages and Systems*, 12(2):303–339, April 1990.

[3] Toshiro Araki and Nobuki Tokura. Flow languages equal recursively enumerable languages. *Acta Informatica*, 15:209–217, 1981.

[4] E. Ashcroft and Z. Manna. Formalization of properties of parallel programs. *Machine Intelligence*, 6:17–41, 1971.

[5] Susan 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.

[6] George S. Avrunin, Ugo Buy, and James Corbett. Automatic generation of inequality systems for constrained expression analysis. Technical Report 90-32, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

[7] George S. Avrunin, Ugo A. Buy, and James C. Corbett. Integer programming in the analysis of concurrent systems. In Larsen and Skou [48], pages 92–102.

[8] George S Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated analysis of concurrent systems with the constrained expression toolset. *IEEE Transactions on Software Engineering*, 17(11):1204–1222, November 1991.

[9] George S. Avrunin, Ugo A. Buy, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Experiments with an improved constrained expression toolset. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, October 1991. ACM SIGSOFT, Association for Computing Machinery.

[10] George S. Avrunin, James C. Corbett, Laura K. Dillon, and Jack C. Wileden. Automated constrained expression analysis of real-time software. Submitted for publication. Available as Constrained Expression Memorandum 91-3., December 1990.

[11] George S. Avrunin, Laura K. Dillon, and Jack C. Wileden. Constrained expression analysis of real-time systems. Technical Report 89-50, Department of Computer and Information Science, University of Massachusetts, 1989.

[12] Roland C. Backhouse. *Program Construction and Verification.* Prentice-Hall, Englewood Cliffs, New Jersey, 1986.

[13] Peter C. Bates and Jack C. Wileden. High-level debugging of distributed systems: The behavioral abstraction approach. *Journal of Systems and Software*, 3:255–264, 1983.

[14] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, 1985.

[15] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic model checking: 10^{20} states and beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 428–439, 1990.

[16] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[17] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model checking and abstraction. In *Proceedings of the 19th ACM Symposium on Principles of Programming Languages*, pages 343–354, January 1992.

[18] E.M. Clarke, D.E. Long, and K.L. McMillan. Compositional model checking. In *Proceedings of the Fourth Annual IEEE Symposium on Logic in Computer Science*, 1989.

[19] Rance Cleaveland, Joachim Parrow, and Bernhard Steffen. The Concurrency Workbench. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 24–37, June 1989. Appeared as *Lecture Notes in Computer Science* 407.

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

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

[22] James C. Corbett. New and improved inequality generation in the constrained expression toolset. Constrained Expression Memorandum 92-1, Department of Computer and Information Science, University of Massachusetts, Amherst, 1992.

[23] James C. Corbett. Verifying general safety and liveness properties with integer programming. In *Proceedings of the Fourth Workshop on Computer Aided Verification*, pages 337–348, 1992.

[24] James C. Corbett and G. Allyn Polk. A tool for generating behaviors in constrained expression analysis. Constrained Expression Memorandum 90-3, Department of Computer and Information Science, University of Massachusetts, Amherst, 1990.

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

[26] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(1):80–86, 1985.

[27] J.W. Davies, D.M. Jackson, G.M. Reed, A.W. Roscoe, and S.A. Schneider. Communication and correctness in timed CSP. Technical report, Programming Research Group Report, Oxford University Computing Laboratory, 1990.

[28] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

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

[30] Laura K. Dillon. Verifying general safety properties of Ada tasking programs. *IEEE Transactions on Software Engineering*, 16(1):51–63, 1990.

[31] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[32] Donald L. Fisher, Donna Saisi, and William M. Goldstein. Stochastic PERT networks: OP diagrams, critical paths, and the project completion time. *Computers & Operations Research*, 12(5):471–482, 1985.

[33] R. Floyd. Assigning meanings to programs. In *Symposium on Applied Mathematics*, volume 19 of Mathematical Aspects of Computer Science, J. Schwartz, editor, American Mathematical Society, New York, pages 19–32, 1967.

[34] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[35] C. Ghezzi, D. Mandriolli, and A. Morzenti. Trio: A logic language for executable specifications of real-time systems. *Journal of Systems and Software*, 12(2):107–123, May 1990.

[36] Patrice Godefroid and Pierre Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. In Larsen and Skou [48], pages 332–242.

[37] David Gries. *The Science of Programming.* Springer-Verlag, New York, 1981.

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

[39] Matthew Hennessey. *Algebraic Theory of Processes.* The MIT Press, Cambridge, Massachusetts, 1988.

[40] C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[41] C. A. R. Hoare. *Communicating Sequential Processes.* Prentice-Hall International, 1985.

[42] G.J. Holzmann. An improved reachability analysis technique. *Software: Practice and Experience*, 18(2):137–161, February 1988.

[43] P. Huber, A. M. Jensen, L. O. Jepsen, and K. Jensen. Towards reachability trees for high-level Petri nets. Technical Report DAIMI PB-174, Department of Computer Science, Aarhus University, Aarhus, Denmark, 1985.

[44] Farnam Jahanian and Aloysius Ka-Lau Mok. Safety analysis of timing properties in real-time systems. *IEEE Transactions on Software Engineering*, 12(5):890–904, 1986.

[45] Gerald M. Karam and Raymond J. Buhr. Starvation and critical race analyzers for Ada. *IEEE Transactions on Software Engineering*, 16(8):829–843, 1990.

[46] Richard A. Kemmerer, editor. *Proceedings of the ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (TAV3).* ACM, New York, Key West, Florida, December 1989. Appeared as *Software Engineering Notes*, 14(8).

[47] Leslie Lamport. A new approach to proving the correctness of multiprocess programs. *ACM Transactions on Programming Languages and Systems*, 1(1):84–97, July 1979.

[48] Kim Guldstand Larsen and Arne Skou, editors. *Computer Aided Verification, 3rd International Workshop Proceedings*, volume 575 of *Lecture Notes in Computer Science*, Aalborg, Denmark, July 1991. Springer-Verlag.

[49] T. J. Leblanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, C-36(4):471–482, April 1987.

[50] Orna Lichtenstein and Amir Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *Proceedings of the Twelfth ACM Symposium on the Priciples of Programming Languages*, pages 97–105, 1985.

[51] Douglas L. Long and Lori A. Clarke. Task interaction graphs for concurrency analysis. In *Proceedings of the 11th International Conference on Software Engineering*, pages 44–52, Pittsburgh, PA, May 1989.

[52] N. A. Lynch and H. Attiya. Using mappings to prove timing properties. In *Proceedings of the ACM Symposium on the Principles of Distributed Computing*, pages 265–280, 1990.

[53] Stephen P. Masticola and Barbara G. Ryder. Static infinite wait anomaly detection in polynomial time. In *Proceedings of the 1990 International Conference on Parallel Processing*, volume II, pages 78–87, 1990.

[54] Charles E. McDowell. A practical algorithm for static analysis of parallel programs. *Journal of Parallel and Distributed Processing*, 6(3):515–536, June 1989.

[55] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer-Verlag, Berlin, 1980.

[56] Robin Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.

[57] Steven S. Muchnick and Neil D. Jones. *Program Flow Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[58] Tadao Murata, B. Shenker, and S. M. Shatz. Detection of Ada static deadlocks using Petri net invariants. *IEEE Transactions on Software Engineering*, 15(3):314–326, 1989.

[59] Lee Osterweil, editor. *Proceedings of the Second Workshop on Software Testing, Analysis and Verification*. ACM, New York, Banff, Canada, July 1988.

[60] J. S. Ostroff. Deciding properties of timed transition models. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):170–183, 1990.

[61] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica*, 6(4):319–340, 1976.

[62] Amir Pnueli. Applications of temporal logic to the specification and verification of reactive systems: a survey of curent trends. In J.W. de Bakker, editor, *Current Trends in Concurrency*, volume 224 of *Lecture Notes in Computer Science*, pages 510–584. Springer-Verlag, 1985.

[63] David K. Probst and Hon F. Lee. Using partial-order semantics to avoid the state explosion problem in asynchronous systems. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 15–24, Providence, RI, 1991. American Mathematical Society.

[64] P. Puschner and Ch. Koza. Calculating the maximum execution time of real-time programs. *Journal of Real-Time Systems*, 1:159–176, 1989.

[65] Marko Rauhamma. A comparative study of methods for efficient reachability analysis. Technical Report Series A, Number 14, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Otaniemi, Otakaari 5 A, SF-02150 ESPOO 15, FINLAND, September 1990.

[66] R. R. Razouk and M. M. Gorlick. A real-time interval logic for reasoning about executions of real-time programs. In *Proceedings of the Third ACM Symposium on Software Testing, Analysis, and Verification*, pages 10–19, 1989.

[67] J. Reif and S. Smolka. The complexity of reachability in distributed communicating processes. *Acta Informatica*, 25(3):333–354, 1988.

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

[69] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7):875–889, 1989.

[70] Kuo-Chung Tai and Evelyn E. Obaid. Reproducible testing of Ada tasking programs. In *Proceedings of the 2nd International Conference on Ada Applications and Environments*, pages 69–79, April 1986.

[71] Richard N. Taylor. Complexity of analyzing the synchronization structure of concurrent programs. *Acta Informatica*, 19:57–84, 1983.

[72] Richard N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26(5):362–376, May 1983.

[73] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B. MIT Press/Elsevier, 1990.

[74] S. Tu, S. M. Shatz, and T. Murata. Theory and application of Petri net reduction for Ada-tasking deadlock analysis. Technical Report 91-15, EECS Department, University of Illinois, Chicago, 1991.

[75] A. Valmari. Error detection by reduced reachability graph generation. In *Proceedings of the Ninth European Workshop on the Application and Theory of Petri Nets*, pages 95–112, Venice, Italy, 1988.

[76] Antti Valmari. Compositional state space generation. In *European Conference on Petri Nets*, pages 43–62, 1990.

[77] Antti Valmari. A stubborn attack on state explosion. In E. M. Clarke and R. P. Kurshan, editors, *Computer-Aided Verification '90*, number 3 in DIMACS Series in Discrete Mathematics and Theoretical Computer Science, pages 25–41, Providence, RI, 1991. American Mathematical Society.

[78] Jack C. Wileden. *Modelling Parallel Systems with Dynamic Structure.* PhD thesis, University of Michigan, 1978.

[79] Wei Jen Yeh and Michal Young. Compositional reachability analysis using process algebra. In *Proceedings of the Symposium on Testing, Analysis, and Verification (TAV4)*, pages 178–187, New York, October 1991. ACM SIGSOFT, Association for Computing Machinery.

[80] Michal Young and Richard N. Taylor. Combining static concurrency analysis with symbolic execution. *IEEE Transactions on Software Engineering*, 14(10):1499–1511, October 1988.

[81] Michal Young, Richard N. Taylor, Kari Forester, and Debra Brodbeck. Integrated concurrency analysis in a software development environment. In Richard 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).

[82] Han Zuidweg. Verification by abstraction and bisimulation. In J. Sifakis, editor, *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*, pages 105–166, June 1989. Appeared as *Lecture Notes in Computer Science* 407.