MIT/LCS/TM-158

SEMAPHORE PRIMITIVES AND STARVATION-FREE MUTUAL EXCLUSION

Eugene William Stark

March 1980

Semaphore Primitives and Starvation-Free Mutual Exclusion

by

Eugene William Stark

January, 1980

Massachusetts Institute of Technology
Laboratory for Computer Science
Cambridge, Massachusetts

Semaphore Primitives and Starvation-Free Mutual Exclusion

by

Eugene William Stark

Submitted to the Department of Electrical Engineering and Computer Science on January 18, 1980 in partial fulfillment of the requirements for the degree of Master of Science

Abstract

Most discussions of semaphore primitives in the literature provide only an informal description of their behavior, rather than a more precise definition. These informal descriptions may be incorrect, incomplete, or subject to misinterpretation. As a result, the literature actually contains several different definitions of the semaphore primitives. The differences are important, since the particular choice of definition can affect whether a solution to the mutal exclusion problem using semaphore primitives allows the possibility of process *starvation*. This thesis attempts to alleviate some of the confusion by giving precise definitions of two varieties of semaphore primitives; here called *weak* and *blocked-set* primitives. It is then shown that under certain natural conditions, although it is possible to implement starvation-free mutual exclusion with blocked-set semaphores, it is not possible to do so with weak semaphores. Thus weak semaphores are strictly less "powerful" than blocked-set semaphores.

Thesis Supervisor: Irene Greif
Title: Assistant Professor of Computer Science

Key Words: parallel processes, mutual exclusion, semaphores, synchronization

Acknowledgements

I am indebted to Carl Seaquist for many productive discussions during the early stages of this work, and for discovering a key idea in the proof of Theorem 3.14. I would like to thank my thesis supervisor, Professor Irene Greif, for her patience in reading drafts of this thesis. Thanks also to Russ Atkinson, Jeannette Wing, and especially Jeff Jaffe and Craig Schaffert, for reading and commenting on parts of this thesis at various stages along the way. Finally, I would like to thank the National Science Foundation for providing fellowship support over most of the period during which this research was conducted.

Table of Contents

1. Introduction

When a number of concurrently-executing sequential processes share data, it is necessary to provide some mechanism for synchronizing the processes' accesses to this data, so that harmonious cooperation may be achieved. A large number of *synchronization primitives*, or programming language constructs for expressing synchronization, have been proposed and studied in the extensive literature that has developed on this subject. The complexity of such synchronization primitives covers a wide range: from low-level primitives such as indivisible fetch and store operations on shared memory locations; through more powerful, but still rather low-level primitives such as *semaphores* [DIJKS68] [DIJKS71], and *conditional critical regions* [BRIN72a]; to constructs capable of expressing synchronization at a higher level, for example *monitors* [BRIN72b], or *path expressions* [HABER75]. Each of these constructs has been shown to be useful in solving various *synchronization problems*, such as the "mutual exclusion problem", [DIJKS65] [DIJKS68], the "readers/writers" problems [COURT71], and "bounded buffer" problems like that in [HABER72], to name just a few.

The earliest, and perhaps the most extensively studied of these synchronization problems is the *mutual exclusion problem* [DIJKS65]. This is the problem of ensuring that the execution of certain *critical regions* of code in one sequential process is not interrupted by the execution of similar regions of code in another process. Critical regions are useful, for example, in a timesharing operating system when it is desired to manipulate system queues. These manipulations must not be interrupted when the queue pointers are in an inconsistent state, or disaster may result. It is possible to produce this mutual exclusion of critical regions using indivisible fetch and store operations to shared memory locations as the only mechanism for synchronizing between processes; and quite a number of solutions based on this principle, for example [DIJKS65] [KNUTH66] [LAMPO74], have been devised. These solutions have the common property that processes synchronize via "busy waiting"; that is, when two processes desire to execute in their critical regions simultaneously, one of the processes waits by looping and testing shared memory locations until the other process has completed execution in its critical region. Such solutions are therefore wasteful in the sense that processor time is used for waiting, instead of for more useful computation.

As a way of improving upon this situation, and of simplifying the solution of the mutual exclusion problem, Dijkstra [DIJKS68] proposed the use of *semaphores*. A semaphore is a special type of variable, shared between processes, that may be manipulated only by two special operations, designated **P** and **V**. A semaphore variable may take on only nonnegative integer values. One definition of the effect of the semaphore operations is as follows: A process performing a **P** operation on a semaphore variable s tests the value of s to see if it is greater than zero. If so, then s is decremented and the process proceeds. The test and resultant decrement are performed in one indivisible step. If the value of s is not greater than zero, the process is said to become *blocked* on the semaphore s, and must wait to be signalled by some process executing a $\mathbf{V}(s)$ operation. A process executing a $\mathbf{V}(s)$ operation checks to see if there are any processes blocked on s. If there are blocked processes, then one of them is signalled and allowed to proceed. If there are no blocked processes, then s is simply incremented. The $\mathbf{V}(s)$ operation is assumed to be performed in a single indivisible step. This definition appears to be the one intended by Dijkstra in [DIJKS68], however, as we shall see, there are other, competing definitions of the **P** and **V** primitives. Which definition is used affects subtle properties of solutions to the mutual exclusion problem implemented with semaphores.

One such subtle property is whether a solution allows the possibility of *starvation* of one or more processes. Starvation occurs when one or more processes attempting to enter their critical regions are delayed forever by a continuous stream of other processes executing critical regions. The goal of this thesis is to investigate how various definitions of the semaphore operations affect whether solutions to the mutual exclusion problem implemented with semaphores allow the possibility of starvation.

1.1 Various Informal Definitions of Semaphores

There has been a good deal of confusion in the literature regarding the definitions of semaphore operations. Much of this has resulted from the fact that often the effect of the semaphore **P** and **V** operations is merely described informally, rather than specified precisely. These informal definitions may be incorrect, incomplete, or subject to misinterpretation. In this section, several different definitions of semaphores will be informally introduced. However, it will not be possible to use these informal definitions in our formal investigation. The informal definitions presented in this section are therefore intended only to serve as motivation for their formal counterparts to be given in Chapter 2.

1.1.1 Binary Versus General Semaphores

We will distinguish, as is done by Dijkstra in [DIJKS68], between *binary* and *general* semaphores. The discussion above gave a definition of *general* semaphores. *Binary* semaphores are similar to general semaphores, except that the binary semaphore variable may take on only the values zero and one. The effect of a binary **P** operation on a semaphore variable *s* is identical to that of a general **P** operation. However, to ensure that the value of the variable *s* never exceeds one, a binary **V**(*s*) operation will simply set *s* to one, rather than incrementing *s* as is done in a general **V**(*s*) operation. Note that if the value of a binary semaphore variable *s* is one, which implies that there are no processes blocked on *s*, then execution of a **V**(*s*) has no effect.

1.1.2 Blocked-Set and Blocked-Queue Semaphores

Consider Dijkstra's definition of general semaphores presented above. Processes that are blocked within a **P** operation on a semaphore variable *s* are distinguished from processes that have not yet examined the value of the variable *s* and hence have not yet become blocked, in that the execution of a **V**(*s*) will cause a blocked process to be selected in preference to a process that is not blocked. However, all blocked processes are treated equally as far as being selected is concerned; no effort is made to distinguish processes that have been blocked for a short length of time from those that have been blocked for a longer period. The group of blocked processes at any instant of time may therefore be modeled as a *set*, from which a **V** operation chooses at random a process to be signalled. Let us call semaphores with this type of blocking discipline *blocked-set* semaphores. It is also possible to define *blocked-queue* semaphores, which are like blocked-set semaphores except that the group of blocked processes is maintained as a *queue*, instead of as a set. Processes becoming blocked are placed at the end of the queue, and processes are selected for signalling from the head of the queue. Both binary and general blocked-set and blocked-queue semaphores may be defined.

1.1.3 Weak Semaphores

As was mentioned above, it appears that blocked-set semaphores are the type that Dijkstra intended to define in [DIJKS68]. In [DIJKS71] he indicates the possibility of defining blocked-queue semaphores as well. Blocked-set semaphores also appear to be the type used in [COURT71], [COURT72], [HABER72], and [LIPTO73]. However, a third type of semaphore, much different than either blocked-set or blocked-queue semaphores, is also found in the literature. This is the type of semaphore that may be implemented with indivisible "test-and-set" instructions as follows: A process attempting to perform a **P** operation on a semaphore variable *s* executes a busy-waiting loop in which the value of *s* is continually tested. As soon as *s* is discovered to have a value greater than zero, it is decremented; the decrement and the immediately preceding test being performed as one indivisible step. A **V**(*s*) operation simply increments *s* in an indivisible step. We will call this type of semaphore a *weak* semaphore. The preceding definition is for weak general semaphores, although it is possible to define weak binary semaphores as well. One of the results we will prove is that weak semaphores are indeed significantly "weaker" than blocked-set or blocked-queue semaphores, when their starvation properties are considered.

Definitions of weak semaphores equivalent to the definition above may be found in [PRESS75], [SHAW74], [KOSAR73]. However, it is here that confusion over the definitions of the semaphore operations becomes evident. The definitions given in [PRESS75] are at best incomplete. Under the most straightforward interpretation though, they appear to define weak semaphores. Haberman [HABER76] criticizes [PRESS75] for this, and for presenting a definition of weak semaphores without indicating the difference between this version and the blocked-set semaphores defined by Dijkstra.

Shaw [SHAW74] first presents a definition of semaphores like that of blocked-set semaphores given above, but then proceeds to show (incorrectly) how this type of semaphores may be "implemented" with test-and-set instructions as primitives. What actually results from this implementation is weak semaphores. In addition, he shows how general semaphores may be simulated using binary semaphores. With the test-and-set implementation of binary semaphores, this simulation is incorrect. Kosaraju [KOSAR73] points out the difference between weak and blocked-set semaphores, and indicates the importance of this distinction; however he believes Dijkstra's definition in [DIJKS68] to define weak, rather than blocked-set semaphores. Although Dijkstra's definition is rather vague, this belief appears ill-founded, in view of the agreement between [DIJKS68], [DIJKS71], [COURT71], [COURT72], [HABER72], and [HABER76].

The behavior of the weak semaphore operations is most easily described using the "busy-waiting" implementation, as was done above. However, it may seem unreasonable that busy-waiting is used in the implementation of semaphores, which are introduced partly because of the desire to avoid busy-waiting. It is also possible to imagine non-busy-waiting implementations that produce the same abstract synchronization behavior. For example, suppose that all processes are run in time-shared fashion on a single processor. When a process initiates a **P**(s) operation and discovers that the value of s is zero, it places itself in a list of processes waiting on the semaphore s, and releases the processor. The system scheduler will not reschedule processes waiting on s as long as s has the value zero. A process performing a **V**(s) simply increments s, however rescheduling does *not* necessarily occur after the incrementation.

We assert that the two informal definitions of weak semaphores presented above, and the model of weak semaphores to be introduced in the next chapter, define "equivalent" synchronization behaviors, that are essentially different from the behavior of blocked-set and blocked-queue semaphores. This statement will not be proved, since a suitable definition of "equivalence" is lacking, and it is beyond the scope of this thesis to develop one. However, [KELLE76], [DOEPP76], and [KWONG78] indicate one way in which this might be done, with their discussions of "abstract" semaphores and "implementations" of this abstract behavior.

1.1.4 Yet Another Type of Semaphores

The definition of semaphores given in [COFFM73] p. 68, while using busy-waiting to implement a kind of weak semaphore, is somewhat different from the other definitions discussed above in the following sense: His semaphore variables may take on negative, as well as nonnegative, integer values. This in itself would not be such an important difference, were it not for the way the negative values are used to control processes performing a **P** operation. A process performing a **P** operation first decrements the semaphore variable. It then loops until the value of the variable is nonnegative. A **V** operation simply increments the value of the semaphore variable. Therefore, if several processes attempt to perform **P** operations, and cause the value of the semaphore variable to go well below zero, none of the processes will be able to proceed until sufficient **V** operations have been performed to make the value of the semaphore variable at least zero. At this point, all processes that were waiting will be allowed to proceed. This is in contrast to the other definitions presented above, where a **V** operation immediately allowed one waiting process to proceed. We will not consider the type of semaphores defined in [COFFM73].

As a final indication of the confusion in the literature on semaphores, it is interesting to note that all the sources above claim that their definitions define "Dijkstra's semaphore primitives".

1.2 Starvation Properties of the Various Definitions

The weak, blocked-set, and blocked-queue semaphore primitives defined above have different starvation properties. To see why this might be true, let us see what happens when each definition is used in a simple attempt to solve the mutual exclusion problem. Consider a number of processes, each executing the following program:

```
      semaphore s initially 1;

loop: <noncritical region>
      P(s);
      <critical region>
      V(s);
      goto loop;
```

Each process continually alternates execution between its *critical region* and its *noncritical region*. In order to ensure that mutual exclusion of critical regions among all the processes is obtained, the critical region is bracketed by a **P**(*s*)-**V**(*s*) pair. Since the value of the semaphore variable *s* is initially one, and a process desiring to enter the critical region must first perform a **P**(*s*) operation, whenever some process is in its critical region, the value of *s* is zero. Hence other processes attempting to perform **P** operations and enter their own critical regions must wait. Mutual exclusion is therefore obtained regardless of whether weak, blocked-set, or blocked-queue semaphores are used.

Suppose that the semaphore operations are of the weak variety, and consider the execution of two processes, process 1 and process 2. Suppose that process 1 finds the value of s to be one, and proceeds into its critical region. Since the value of s is now zero, process 2 is unable to complete its $\mathbf{P}(s)$ operation, and therefore waits within the **P** operation for the value of s to become positive. Now suppose that process 1 completes execution in its critical region, and performs the $\mathbf{V}(s)$ operation, setting s to one. Since we have assumed the semaphore operations to be weak, process 2 does not complete its $\mathbf{P}(s)$ operation immediately, but must retest the semaphore variable s (or be rescheduled). It is possible, if process 1 executes quickly enough, for it to loop around and perform another $\mathbf{P}(s)$ operation, resetting s to zero, *before process 2 could get around to noticing that s ever had the value one.* This scenario may continue indefinitely, with the result that process 2 "starves" forever within its $\mathbf{P}(s)$ operation. Note that this argument relies on the fact that, in determining the behavior of a system of concurrent processes, we may make no assumptions about the relative speeds of the processes, and must consider all possible orders of executions of steps of the processes as equally likely.

Now, suppose instead that the semaphore operations are defined to be blocked-set operations. The scenario described in the preceding paragraph is no longer possible, since the execution of a **V** operation by process 1 immediately causes process 2 to complete its $\mathbf{P}(s)$ operation. Since s is never set to one, it is not possible for process 1 to complete another $\mathbf{P}(s)$ before process 2 finishes its critical region and performs a $\mathbf{V}(s)$. However, although starvation is no longer possible with two processes, with three or more processes it again becomes possible for a process to wait forever within the $\mathbf{P}(s)$ while other processes successfully complete infinitely many $\mathbf{P}(s)$ operations. The reason for this is that the blocked-set **V** operation selects the blocked process to signal at random, and in particular, gives no preference to a process that may have been blocked for a long time. This situation may be alleviated if blocked-queue semaphores are used. Blocked-queue semaphores impose a FIFO discipline on the blocked processes, and hence any blocked process will be allowed to proceed after a number of **V** operations that is at most equal to the number of processes in the system.

1.3 Relative "Power" of the Types of Semaphores

The simple scenario just presented indicates that, although weak, blocked-set, and blocked-queue semaphores are all able to implement mutual exclusion of critical regions, the three types of semaphores are evidently not equivalent if the possibility of starvation is taken into consideration. We will be interested in obtaining more detailed information of this type about the relative "power" of the various kinds of semaphore primitives. We will obtain this information by posing and answering questions like: "Is it possible to implement starvation-free mutual exclusion with a given kind of semaphore?" If the answer to this question is "Yes", then, "Can natural constraints be imposed under which it is no longer possible to implement starvation-free mutual exclusion?" and "Is it possible to distinguish between the relative 'power' of the various definitions of semaphores on the basis of the strength of these constraints?" It turns out to be trivially possible to implement starvation-free mutual exclusion with either blocked-queue binary or blocked-queue general semaphores, under any of the constraints we will impose. We will therefore concentrate our efforts on determining the differences in "power" between weak binary, weak general, blocked-set binary, and blocked-set general semaphores.

The "flavor" of this investigation is similar to that of [BURNS79], where solutions to the mutual exclusion problem are studied for a system of processes that synchronize not with semaphore operations, but with a general "test-and-set" operation on a single shared variable. In that study, bounds are obtained on the number of distinct values that this variable must be able to record, if solutions are to exist to the mutual exclusion problem, and to the starvation-free mutual exclusion problem.

The most important similarity between our study and that of [BURNS79] is that in both, results are stated and proved asserting the existence or nonexistence of solutions to the mutual exclusion problem satisfying various properties. In both studies, existence results are proved by displaying a solution to the mutual exclusion problem, and proving that it has the stated properties. Results asserting the nonexistence of solutions are proved indirectly, by assuming the existence of a solution satisfying the stated properties, and then inferring the existence of a computation that contradicts one or more assumptions.

Another similar investigation was performed by Miller and Yap [MILLE77]. In that paper, a model of parallel processes is presented, and used to formalize a number of properties "desirable" for a solution to the mutual exclusion problem. Under the assumption that a process may either fetch or store the value of a single shared variable as part of a single indivisible action, they are able to establish some lower bounds on the number of global variables required to implement starvation-free mutual exclusion.

Lipton [LIPTO73] also investigates issues of the expressive "power" of various types of semaphore-like synchronization primitives. However, his focus of study is not restricted to the ability of the various primitives to implement mutual exclusion. Instead, the synchronization primitives are distinguished on the basis of their ability to implement various abstractly specified synchronization behaviors.

1.4 Outline of Thesis

The remainder of the thesis is organized as follows: In Chapter 2 a model of parallel computation called *parallel programs* will be introduced. This model is an adaptation of similar models used by [KELLE76], [DOEPP76], and [KWONG78]. We will need to make some restrictions on this rather general model, to reflect the fact that we are interested in modeling a specific type of parallel computation; that is, a fixed number of sequential, deterministic processes that communicate via shared variables, and that synchonize with semaphore operations. We will call this restricted class of the parallel programs *systems of processes.* Certain systems of processes, which we call *mutual exclusion systems,* model a set of processes that alternate their execution between a critical region and a noncritical region. We will define a *solution to the starvation-free mutual exclusion problem* to be a mutual exclusion system that guarantees certain "desirable" properties, such as mutual exclusion, freedom from deadlock, and freedom from starvation.

In Chapter 3 we will use the formal model presented in Chapter 2 to investigate the relationship between the definition of semaphores and solutions to the starvation-free mutual exclusion problem. The results proved in Chapter 3 will be "negative" results which assert that under certain conditions, solutions to the starvation-free mutual exclusion problem do not exist. In Chapter 4, actual solutions to the starvation-free mutual exclusion problem will be exhibited in support of complementary "positive" or "existence" results.

Chapter 5 and the Appendix are devoted to a correctness proof for what is perhaps the most interesting of the solutions presented in Chapter 4. No other solutions are proved correct, because the length of such proofs would be prohibitive. In Chapter 5, three techniques for proving statements about parallel programs are developed, and then applied in the proof. While the most interesting and intuitive parts of the proof are presented in Chapter 5, the remainder, consisting largely of tedious, mechanical verification, has been left to the Appendix. It is possible to understand the arguments in Chapter 5 without reference to the Appendix. Finally, Chapter 6 contains a summary of results, and suggests possible directions for future investigation.

2. A Definition of Starvation-Free Mutual Exclusion

If we are to make and prove statements that assert the nonexistence of solutions to the starvation-free mutual exclusion problem under various constraints, then we must precisely define the class of possible solutions. For this purpose, we will use a model of parallel computation, called *parallel programs*, which is an adaptation of those used in [KELLE76], [DOEPP76], and [KWONG78]. The parallel program model is capable of modeling much more general classes of parallel computation than will usually concern us, so a restricted class of parallel programs, which we call *systems of processes*, will also be defined. We will further identify certain systems of processes, termed *mutual exclusion systems*, which model a number of processes that repeatedly alternate between a *critical region* and a *noncritical region*. We will then define what it means for a mutual exclusion system to *have the mutual exclusion property*, to be *deadlock-free*, *free from indefinite postponement*, and *starvation-free*. A *solution to the starvation-free mutual exclusion problem* will be defined as a mutual exclusion system that has these properties.

2.1 Parallel Programs

Keller [KELLE76], introduces a model of parallel computation, called a *transition system*, which consists of a set of *states*, and a binary relation on those states, called the *transition relation*. A computation of the system is a sequence of states; each state related to the next in the sequence by the transition relation. Keller uses a graphical notation as a syntax for the definition of specific transition relations. These graphs, which he calls *parallel programs*, are a kind of *labeled Petri nets*. Recall that an ordinary Petri net [HOLT70] is a bipartite directed graph in which the two classes of nodes are called *places* and *transitions*. At any given instant, the places of a Petri net are *marked* with varying numbers of *tokens*. Markings evolve through what is called *firing* the transitions. Briefly, a transition is *enabled* if there is at least one token on each place that is its immediate ancestor. An enabled transition may *fire* by removing one token from each of these ancestor places, and adding one token to each of its immediate descendants.

The parallel programs we will be using are Petri nets whose transition nodes have been *labeled* with an *enabling predicate* and an *action function*. Associated with each net is a set of *variables*. The enabling predicate describes a condition on the values of those variables. For a transition in a parallel program to be enabled, not only must the token placement condition be satisfied, but the enabling predicate for that transition must hold as well. The action function describes the change in the values of the variables that takes place when the transition is fired.

The variables in our parallel programs will take their values in a domain of values *VAL*, which contains the natural numbers, and all finite sets of natural numbers. A *natural number variable* is a variable that always takes on natural number values. Similarly, a *set variable* is a variable whose value is always a finite set of natural numbers. All variables in any parallel program presented in this thesis will be either set variables or natural number variables.

Definition 2.1 - A *parallel program* Γ is a five-tuple, $(\mathcal{V}_\Gamma, \mathcal{X}_\Gamma, \mathcal{Q}_\Gamma, q_I, label_\Gamma)$, where:

(1) $\mathcal{V}_\Gamma$ is a finite set of variable names.

(2) $\mathcal{X}_\Gamma = (\mathcal{P}_\Gamma, \mathcal{T}_\Gamma, \mathcal{A}_\Gamma)$ is a bipartite directed graph in which $\mathcal{P}_\Gamma$, the set of *places*, and $\mathcal{T}_\Gamma$, the set of *transitions*, are the two classes

of nodes, and $\mathcal{A}_\Gamma \subset (\mathcal{P}_\Gamma \times \mathcal{T}_\Gamma) \cup (\mathcal{T}_\Gamma \times \mathcal{P}_\Gamma)$ is the set of arcs.

(3) $\mathcal{Q}_\Gamma$ is the set of *states* of Γ. Each element q of $\mathcal{Q}_\Gamma$ is a mapping that assigns a natural number $q(p)$ to each $p \in \mathcal{P}_\Gamma$, and a value $q(v) \in VAL$ to each $v \in \mathcal{V}_\Gamma$.

(4) The state q_I is a distinguished element of $\mathcal{Q}_\Gamma$ called the *initial state.*

(5) The function $label_\Gamma$ assigns to each $t \in \mathcal{T}_\Gamma$ a pair of functions B_t and F_t, where B_t: $\mathcal{Q}_\Gamma$ ---> {**true**, **false**} is called the *enabling predicate,* and F_t: $\mathcal{Q}_\Gamma$ ---> ($\mathcal{V}_\Gamma$ ---> VAL) is called the *action function* for t. The functions B_t and F_t have the property that if $q, q' \in \mathcal{Q}_\Gamma$, and $q(v) = q'(v)$ for all $v \in \mathcal{V}_\Gamma$, then $B_t(q) = B_t(q')$ and $F_t(q) = F_t(q')$.

If q is a state, p is a place, and $q(p) = k$, then we will say that there are k *tokens* at place p in state q. The latter part of condition (5) above states that enabling predicates and action functions depend only upon values of variables, and not upon the placement of tokens.

If Γ is a parallel program, and $n \in \mathcal{P}_\Gamma \cup \mathcal{T}_\Gamma$, then define $input(n)$ to be the set of all $n' \in \mathcal{P}_\Gamma \cup \mathcal{T}_\Gamma$ such that $(n', n) \in \mathcal{A}_\Gamma$. Similarly, define $output(n)$ to be the set of all $n' \in \mathcal{P}_\Gamma \cup \mathcal{T}_\Gamma$ such that $(n, n') \in \mathcal{A}_\Gamma$.

Definition 2.2 - Let Γ be a parallel program and suppose that $q \in \mathcal{Q}_\Gamma$ and $t \in \mathcal{T}_\Gamma$. Then the *next state* function $nxt(q, t)$ is defined iff:

(1) $q(p) > 0$, for all $p \in input(t)$

(2) $B_t(q) =$ **true**

If $nxt(q, t)$ is defined, and $q' = nxt(q, t)$, then the following relationships hold between q and q':

(3) $q'(p) = q(p) - 1$ for all $p \in (input(t) - output(t))$

(4) $q'(p) = q(p) + 1$ for all $p \in (output(t) - input(t))$

(5) $q'(p) = q(p)$ for all $p \in input(t) \cap output(t)$ and for all $p \notin input(t) \cup output(t)$

(6) $q'(v) = (F_t(q))(v)$ for all $v \in \mathcal{V}_\Gamma$

If α is a finite sequence of transitions, then $nxt(q, \alpha)$ is defined by composition in the obvious fashion.

If $nxt(q, t)$ is defined, then we say that t is *enabled* in state q, and that $q' = nxt(q, t)$ is the result of *firing* transition t in state q.

Definition 2.3 - Let Γ be a parallel program, and suppose that $q_0 \in \mathcal{Q}_\Gamma$. A finite or infinite sequence of transitions $\alpha = t_0 t_1 \ldots$ is called an *execution sequence from* q_0 *for* Γ, *with corresponding state sequence* $q_0 q_1 \ldots$, if for each $i \geq 0$, transition t_i is enabled in state q_i, and $q_{i+1} = nxt(q_i, t_i)$. A state q' is *reachable* from a state q if there is an execution sequence β such that $q' = nxt(q, \beta)$. An *initial* execution sequence is an execution sequence from q_I.

Definition 2.4 - Let Γ be a parallel program. We say a transition t *affects* a variable v if there exists a state q such that $(F_t(q))(v) \neq q(v)$. We say that t *depends on* v if there exist states q and q', with q' identical to q except for the value assigned to v, such that either $B_t(q) \neq B_t(q')$ or $(F_t(q))(v') \neq (F_t(q'))(v')$ for some variable v' affected by t.

A pictorial representation of a parallel program is shown in Figure 2.1. Places are denoted by circles, and transitions by horizontal bars. Each transition t is labeled by a statement of the form **when** B **do** $(v_1, \ldots, v_m) := F$, where B is a truth-valued expression describing the enabling predicate B_t, and the multiple assignment $(v_1, \ldots, v_m) := F$ describes the action function F_t. We will follow the convention that if t affects no variables, then the **do** part of the label of t will be omitted, and if B_t is identically true, then the **when** part will be omitted, unless this would result in a completely unlabeled transition.

Keller shows how parallel programs may be used to model parallelism, nondeterminism, and processes that *fork* and *join*. It is particularly convenient to model the instruction counters of concurrently executing processes by positions of tokens on the parallel program graphs, and we will do this, in fact, in the next section.

2.2 Systems of Processes

For our purposes, it will be convenient to introduce some syntactic restrictions on parallel programs. The reason for this is that we will be studying a rather specific type of parallel computation: systems of a fixed number of sequential, deterministic processes, which communicate via shared global variables, and synchronize with semaphore operations. This restricted class of parallel programs will be called *systems of processes*. Before giving the definition, though, some motivational discussion will be helpful.

A system of N processes will be a parallel program whose graph consists of N disconnected subgraphs. The ith subgraph is called the *graph for process i*. In any reachable state, there will be exactly one token on the graph for process i, modeling the position of the instruction counter for the ith process. To ensure that any reachable state has a meaningful configuration of tokens, we require that in the

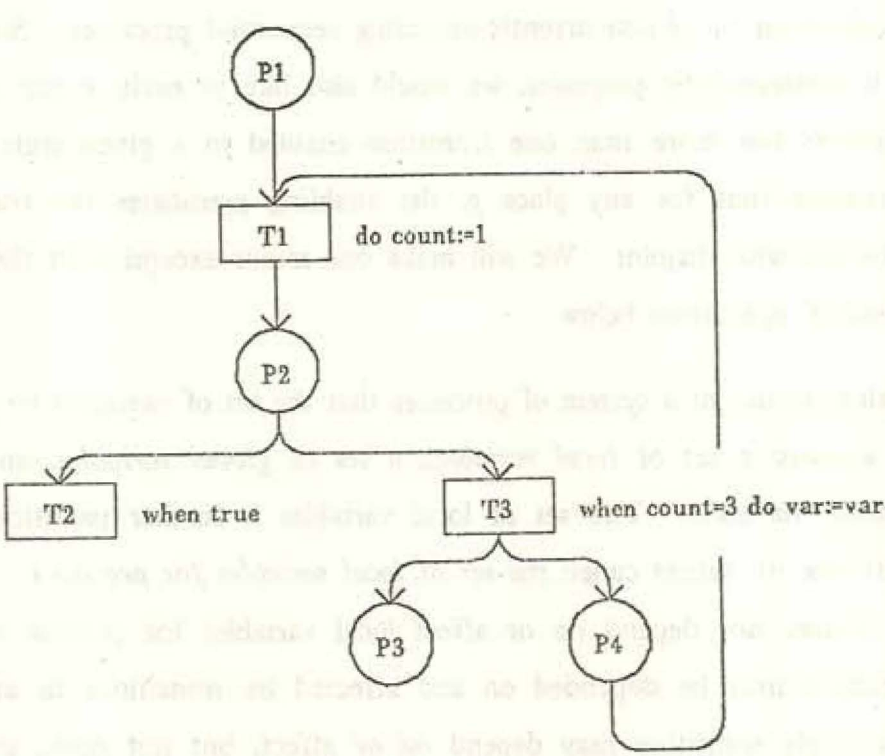


Figure 2.1 - Example of a Parallel Program

initial state q_I there be exactly one token in the graph for each process, and that each transition have exactly one input place and one output place. This means that firing a transition preserves the total number of tokens in the graph. Since graphs for distinct processes are disconnected, tokens cannot "cross over" from one process to another, and hence firing transitions preserves the single token in the graph for each process as well.

The restrictions outlined so far ensure that a system of processes accurately models a collection of N concurrently-executing sequential processes. Since we are interested in deterministic processes, we would also like to exclude the situation in which a process has more than one transition enabled in a given state. We will therefore require that for any place p, the enabling predicates for transitions in $output(p)$ be pairwise disjoint. We will make one minor exception in the definition of blocked-set **V** operations below.

We also require of a system of processes that the set of variables be partitioned into three subsets: a set of *local variables*, a set of *global variables*, and a set of *synchronization variables*. The set of local variables is further partitioned into N subsets, with the ith subset called the set of *local variables for process i*. Transitions in process j may not depend on or affect local variables for process i, if $j \neq i$. Global variables may be depended on and affected by transitions in any process, however, a single transition may depend on or affect, but not both, at most one global variable. This models processes that access shared variables with indivisible fetch and store operations. It is convenient, and results in no loss of generality, to assume that the sets of local variables for any two processes are in one-to-one correspondence.

Certain subgraphs of a system of processes will be identified as modeling *semaphore operations.*

> **Definition 2.5** - Let Γ be a parallel program, and let Ξ be a subgraph of $\mathfrak{X}_\Gamma$. We will call Ξ a *semaphore operation* on the variable s if the following hold:
>
> (1) Ξ has one of the forms shown in Figure 2.2 or Figure 2.3. The subgraph Ξ will be called a weak/blocked-set, binary/general, **P**/**V** operation, depending upon which graph it matches. The subgraph Ξ will then have a distinguished place, called the *input place,* as indicated in the figures.
>
> (2) Connections with the rest of the graph $\mathfrak{X}_\Gamma$ are made such that all arcs from outside Ξ into Ξ must terminate at the input place in Ξ. There must be no arcs from inside Ξ out except for those shown in the figure, and there must be a unique place not in Ξ, at which all arcs from inside Ξ out terminate.

Note that while transition labels in weak **P**(s) and **V**(s) operations mention only the *semaphore* variable s, blocked-set operations make use of two additional *scheduling* variables, whose names are formed by adjoining *_en* and *_bl* to the name of the semaphore variable; in this case to form "*s_en*" and "*s_bl*". The semaphore variable s is a natural number variable, whereas *s_en* and *s_bl* are set variables. We prohibit any transition that is not in a semaphore operation from affecting or depending on semaphore or scheduling variables.

Note that within the graphs for blocked-set semaphore operations, reference is made to the process number i. It is important to recognize that i does *not* represent a parallel program variable; it only serves to indicate that the labels of transitions of

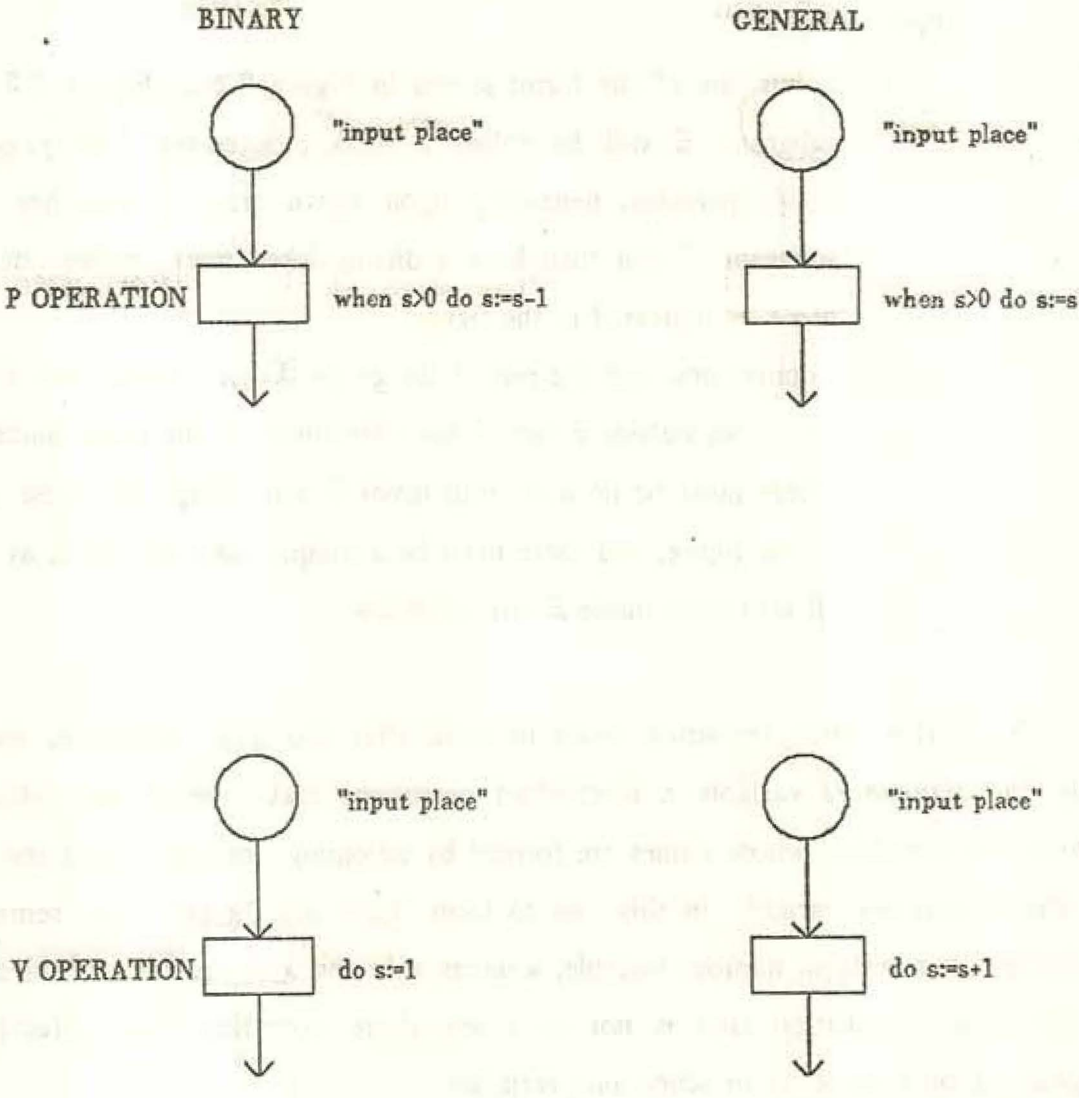
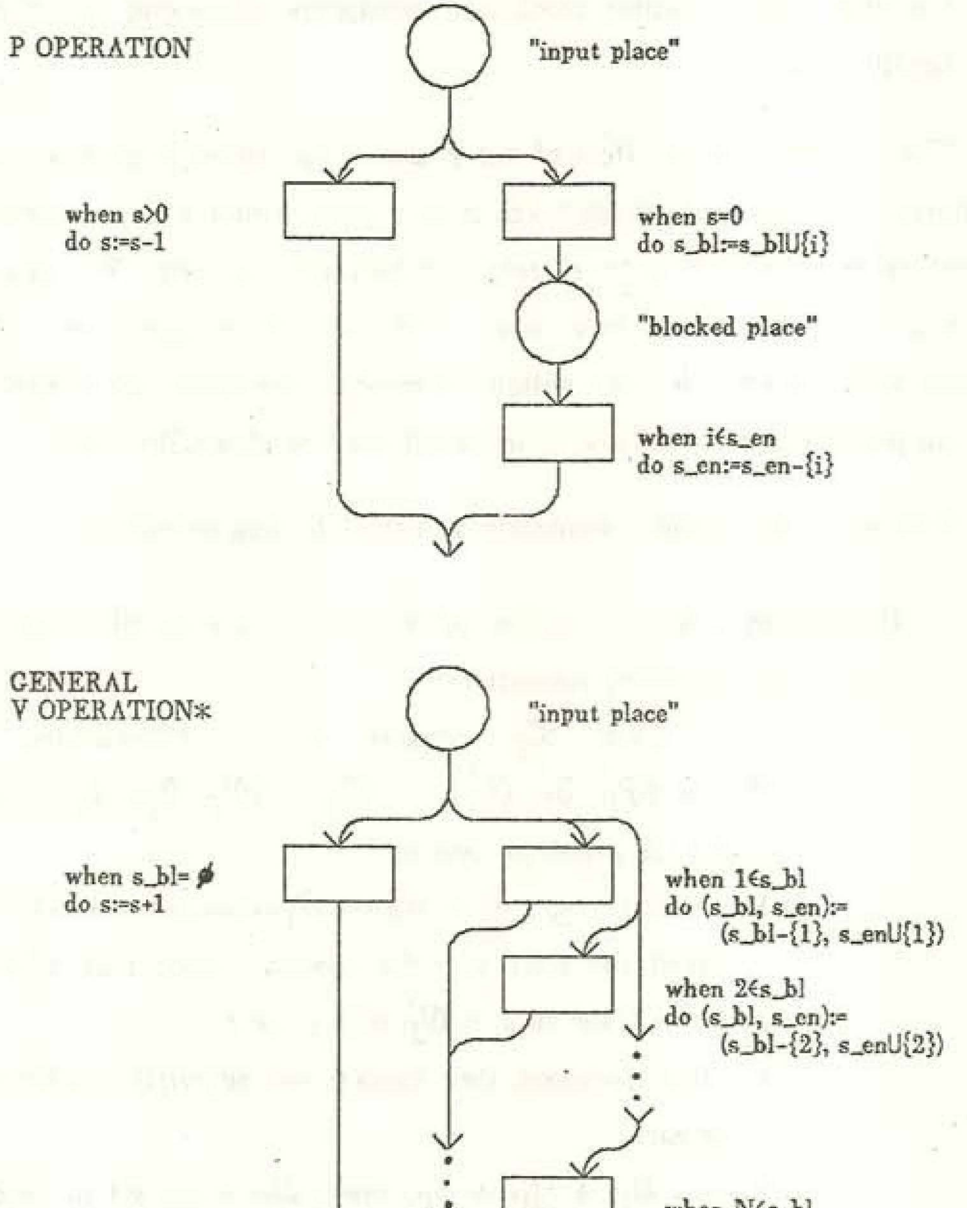


Figure 2.2 - Weak Semaphore Operations



*For binary V operation, replace "do $s := s + 1$" by "do $s := 1$"

Figure 2.3 - Blocked-Set Semaphore Operations

a blocked-set semaphore operation in process 3, for example, must contain the numeral "3" in positions where "*i*" is used in Figure 2.3. The process number is used for scheduling purposes within blocked-set semaphore operations; this is discussed in more detail below.

The synchronization effect of the **P** and **V** operations is obtained through the possibility that a process whose token is at a place within a **P** operation may have no enabled transitions in a given state, and be forced to wait. We wish to require that this type of waiting *only* occur within semaphore operations; therefore we stipulate that if a place p is not within a semaphore operation, the disjunction of the enabling predicates of transitions in *output*(p) must be identically true.

The above discussion is summarized in the following definition:

Definition 2.6 - A *system of N processes* is a parallel program Γ that satisfies the following restrictions:

(1) The graph $\mathcal{X}_\Gamma$ consists of N disconnected subgraphs $\mathcal{X}_\Gamma^1 = (\mathcal{P}_\Gamma^1, \mathcal{T}_\Gamma^1, \mathcal{A}_\Gamma^1), \dots, \mathcal{X}_\Gamma^N = (\mathcal{P}_\Gamma^N, \mathcal{T}_\Gamma^N, \mathcal{A}_\Gamma^N)$, where $\mathcal{X}_\Gamma^i$ is called the *graph for process i*.

(2) For each i, $1 \leq i \leq N$, the set $\mathcal{P}_\Gamma^i$ contains a distinguished place p, called the *start place* for process i, such that $q_I(p) = 1$, and $q_I(p') = 0$ for all $p' \in \mathcal{P}_\Gamma^i$ with $p' \neq p$.

(3) If t is a transition, then *input*(t) and *output*(t) contain exactly one place each.

(4) $\mathcal{V}_\Gamma = \mathcal{L}_\Gamma + \mathcal{G}_\Gamma + \mathcal{S}_\Gamma$, where $\mathcal{L}_\Gamma$ is the set of *local variables*, $\mathcal{G}_\Gamma$ is the set of *global variables*, and $\mathcal{S}_\Gamma$ is the set of *synchronization variables*. (The symbol "+" denotes disjoint union.) The set of synchronization variables is partitioned into the set of *semaphore variables*, and the set of *scheduling variables*.

The set $\mathfrak{L}_\Gamma$ is partitioned into N subsets, $\mathfrak{L}_\Gamma^1, \ldots, \mathfrak{L}_\Gamma^N$, where $\mathfrak{L}_\Gamma^i$ is called the set of *local variables for process i.* There is a one-to-one correspondence between $\mathfrak{L}_\Gamma^i$ and $\mathfrak{L}_\Gamma^j$ for all i and j.

(5) Either all semaphore operations in $\mathfrak{X}_\Gamma$ are weak, or all are blocked-set, in which case we say that Γ *has weak or blocked-set semaphores,* respectively. The process number referred to within each blocked-set semaphore operation must match that of the process in whose graph the semaphore operation appears. If Γ has weak semaphores, then Γ has no scheduling variables. If Γ has blocked-set semaphores, then corresponding to each semaphore variable *s*, are two scheduling variables, *s_en* and *s_bl*. Both *s_en* and *s_bl* are set variables with initial value $\varnothing$.

(6) If $t \in \mathfrak{T}_\Gamma^i$, then t must neither depend on nor affect any variable in $\mathfrak{L}_\Gamma^j$, for all $j \neq i$. If t is not part of a semaphore operation, then t must neither depend on nor affect any synchronization variables. Also, either: (a) no global variables are affected or depended on by t; (b) t depends on a single global variable, but affects no global variables; or (c) t affects a single global variable, but depends on no global variables.

(7) If p is a place not in a semaphore operation, and if $output(p) = \{t_1, \ldots, t_k\}$, with $k > 0$, then the corresponding enabling predicates $B_{t_1}, \ldots, B_{t_k}$ are pairwise disjoint, and $B_{t_1} \vee \ldots \vee B_{t_k}$ is identically true.

We will use superscripts when it is necessary to explicitly indicate the correspondence between the sets $\mathfrak{L}_\Gamma^i$ and $\mathfrak{L}_\Gamma^j$ mentioned in condition (4) above. Thus, if $v^1 \in \mathfrak{L}_\Gamma^1$, then v^j denotes the corresponding variable in $\mathfrak{L}_\Gamma^j$.

2.3 How the Semaphore Operations Work

In this section we will discuss the way in which the graphs of Figure 2.2 and Figure 2.3 model weak and blocked-set semaphore operations.

2.3.1 Weak Semaphores

Figure 2.2 shows the graphs for weak binary and weak general **P** and **V** operations on a semaphore variable *s*. A process whose token arrives at the input place for a weak binary or weak general **P** operation must wait until a state is reached in which the value of *s* is greater than zero. When this occurs, the transition for the **P** operation may fire, decrementing the variable *s*. A process arriving at the input place for a weak **V** operation may proceed without waiting; causing *s* to be set to one in the case of a weak binary **V** operation, or incremented in the case of a weak general **V** operation. Note that there is no queueing or priority mechanism included in the weak semaphore operations. In particular, a process arriving at the input place for a weak **P** operation receives no guarantees about how many processes will complete **P** operations on the same semaphore variable ahead of it.

This definition seems to capture the important properties of the informal definitions of weak semaphores given in Chapter 1. The most important of these is the fact that a process performing a **V**(*s*) operation does not immediately cause a waiting process to complete a **P**(*s*).

2.3.2 Blocked-Set Semaphores

The graphs, shown in Figure 2.3, for binary and general blocked-set **P** and **V** operations on a semaphore *s*, differ from the corresponding graphs for weak semaphores in several important respects. In addition to the semaphore variable *s*, the graphs for the blocked-set operations refer to the scheduling variables *s_en* and *s_bl*. The variable *s_en* is a set variable that represents the set of *enabled* processes for the semaphore s, and the variable s_bl is a set variable that represents the set of processes *blocked* for s.

If the token for process i is at the input place for a blocked-set **P**(s) operation, one of two things can happen, depending upon the value of s. If s is greater than zero, the blocked-set **P** completes exactly as in the case of the weak **P** operation; by decrementing the variable s. However, if the value of s is zero, process i inserts its process number into the blocked set s_bl, and the token for process i advances to the *blocked place.* Process i may not proceed until some other process performing a **V**(s) has transferred the process number i from s_bl to s_en. When this occurs, process i may remove the number i from s_en and proceed.

A process at the input place for a blocked-set **V** operation will do one of two things, depending upon whether or not s_bl is empty. If s_bl is empty, s is set to one in the case of a binary **V** operation, or incremented in the case of a general **V** operation. If s_bl is not empty, some process number j is selected nondeterministically from s_bl, and is transferred to s_en. This has the effect of enabling the blocked process j. The nondeterministic selection is modeled by the N transitions labeled "**when** $j \in s\_bl$ **do** $(s\_bl, s\_en) := (s\_bl-\{ j \}, s\_en\cup\{ j \})$", for $1 \leq j \leq N$; more than one of which may be enabled in any given state, depending upon the contents of s_bl. Note that this is the only exception we will make to our requirement that no process have more than one transition enabled in any state.

This definition of semaphores distinguishes blocked processes, whose tokens are at the blocked place in some **P** operation, from processes that are not blocked. A **V** operation gives blocked processes priority over other processes, however no further distinction is made among blocked processes.

2.4 Mutual Exclusion Systems

We wish to use the system of processes model in our discussion of mutual exclusion. Not all systems of processes model a collection of processes attempting to achieve mutual exclusion of critical regions; therefore we must impose some further restrictions on systems of processes. Systems in the resulting class will be called *mutual exclusion systems.* It will then be possible to formalize the notions of *mutual exclusion, absence of indefinite postponement, freedom from deadlock,* and *freedom from starvation,* as properties "desirable" for a solution to the starvation-free mutual exclusion problem. In fact, we will *define* such a solution to *be* a mutual exclusion system with these desirable properties.

Let us first see what are the important concepts to capture in the definition of a mutual exclusion system. It is typical, in the literature, to present a solution to the mutual exclusion problem as shown in Figure 2.4, where a program to be executed by process *i* is displayed. This particular solution, attributed by Dijkstra to Joseph M. Morris, is interesting in its own right, and will be discussed in detail in Chapter 4.

In the program of Figure 2.4, process *i* executes in an infinite cycle, first performing some computation in the "noncritical region", and then manipulating some variables and semaphores in the "trying region". These manipulations culminate in the entrance of the process into the "critical region". After completing execution in the critical region, more manipulation of variables and semaphores is performed in the "leaving region", and the process returns to the noncritical region to complete the cycle. It is access to the critical region that is being controlled by the "synchronization code" in the trying and leaving region. In this solution, waiting for other processes occurs only in the trying region; however it is possible to imagine solutions in which a process might wait in the leaving region as well.

```
blocked-set general semaphore a, b, m initially 1, 1, 0;
global count1, count2, initially 0, 0;

(Program Executed by Process i)

        loop:       <noncritical region>
                    P(b);
                    $count1 := count1 + 1$;
                    V(b);
                    P(a);
                    P(b);
                    $count1, count2 := count1 - 1, count2 + 1$
                    if $count1 > 0$ then begin
(trying region)             V(b);
                            V(a)
                    end else begin
                            V(b);
                            V(m)
                    end;
                    P(m);
                    <critical region>
                    $count2 := count2 - 1$;
(leaving region)    if $count2 > 0$ then V(m) else V(a);
                    goto loop;
```

Figure 2.4 - Typical Presentation of a Solution to the Mutual Exclusion Problem

The specific computations performed in the critical and noncritical regions are irrelevant, as long as they do not "interfere" with the synchronization protocols of the trying and leaving regions. However, it is usually assumed that critical regions always terminate, although this is not required of noncritical regions. The program of Figure 2.4 can be considered a "solution" to the starvation-free mutual exclusion problem only if it works "correctly", regardless of the order in which processes terminate their noncritical regions.

We will attempt to capture these ideas as follows: A mutual exclusion system will be a system of cyclic processes. Since the specific computation performed in the critical and noncritical regions is irrelevant, these regions may be represented in the mutual exclusion system by single places. We therefore require that each process have two distinguished places, called the *critical place* and the *noncritical place*. The remaining places in the graph are partitioned into the *trying region* and the *leaving region*. Connections are made so that within a process, control always flows from the noncritical place to the trying region, to the critical place, to the leaving region, and back to the noncritical place, in that order. It is convenient to require that each process begin execution with its token at the noncritical place.

For technical reasons, it is convenient to require that the connection between the noncritical place and the trying region be made with a single transition, whose enabling predicate is identically true, and which affects no variables. We also require that the connection between the critical place and the leaving region be made with a similar transition. As a result of these requirements, the graph for process *i* in a mutual exclusion system will have the form shown in Figure 2.5. The place NCP_*i* is the noncritical place for process *i*, CP_*i* is the critical place, and NCT_*i* and CT_*i* are the single transitions connecting the noncritical place with the trying region, and the critical place with the leaving region, respectively. Let us summarize the requirements we have made so far.

> **Definition 2.7** - A *mutual exclusion system of N processes* is a system of *N* processes, where the graph for process *i* has the form shown in Figure 2.5 for $1 \leq i \leq N$. As depicted in Figure 2.5, process *i* has two distinguished places, NCP_*i* and CP_*i*, called the *noncritical place* and *critical place*, respectively. The remainder of the graph for process *i* is divided into the *trying region*, the *leaving region* and two distinguished transitions NCT_*i* and CT_*i*, that connect the noncritical place with the trying region, and the critical place with the leaving region, respectively. We require that the start place in each process be the place NCP_*i*.

If *q* is a state, and $q(\text{CP\_}i) = 1$, then we say that process *i* is *in the critical region* in state *q*. Similarly, we say that process *i* is *in the noncritical region* in state *q* if $q(\text{NCP\_}i) = 1$. If there is a place *p* in the trying region (resp. leaving region) for process *i* such that $q(p) = 1$, then we say that process *i* is *in the trying region* (resp. *in the leaving region*) in state *q*.

Not all execution sequences of a system of processes can be regarded as modeling the execution of "real" processes. In particular, we are only interested in execution sequences that satisfy the so-called *finite delay property*. The finite delay property requires that one process in a system must not run infinitely slower than another (fair scheduling), and appears to be necessary for any discussion of starvation-free synchronization. This property is mentioned by name in [KELLE76], [KWONG78], [MILLE77], and used in the definition of an "admissible schedule" in [BURNS79]. The finite delay property will be incorporated into our model as part of the notion of a "valid" execution sequence.

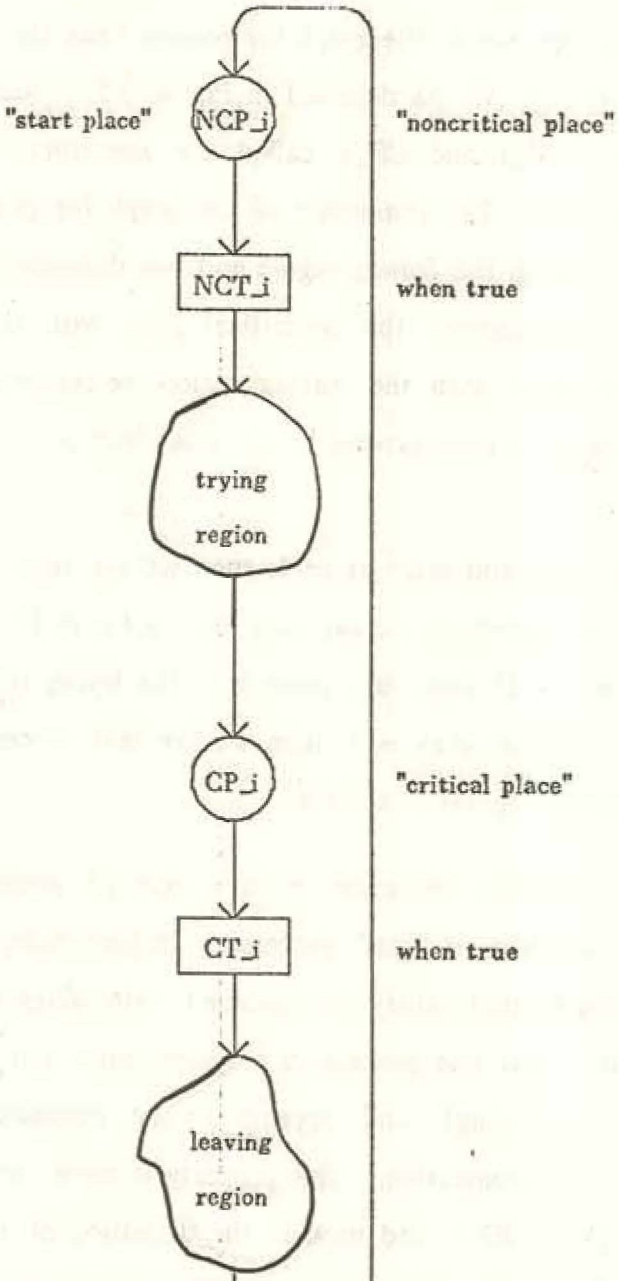


Figure 2.5 - Graph for a Process in a Mutual Exclusion System

Definition 2.8 - Let $\alpha = t_0 t_1 \ldots$ be an execution sequence for a mutual exclusion system of N processes, and suppose that $q_0 q_1 \ldots$ is the corresponding state sequence. Then α is *valid* if it is finite, or for all i, $1 \leq i \leq N$, either

(1) for infinitely many $j \geq 0$, t_j is a transition for process i; or

(2) for infinitely many $j \geq 0$, process i is not enabled in state q_j ; or

(3) there exists $k \geq 0$ such that for all $j \geq k$, process i is in the noncritical region in state q_j

We will say that a state sequence is *valid* if it is the state sequence corresponding to a valid execution sequence.

Stated another way, for an infinite execution sequence to be valid, there can be no process in either the trying, critical, or leaving region, which is enabled for infinitely many *consecutive* states, but fires no transitions. One implication of this is that a process whose token is at the input place for a weak **P** operation can starve if there are infinitely many states in which the value of the semaphore is zero. Another version of the finite delay property, which requires that a process fire transitions even if that process is only enabled *infinitely often*, is discussed in [KWONG78]. Kwong shows that if this stronger version of the finite delay property is adopted then the possibility of starvation is eliminated in many situations. For example, a process at the input place for a weak **P** operation would always be required eventually to complete the **P** operation unless the value of the semaphore variable were zero for infinitely many consecutive states. This means that starvation-free mutual exclusion could be trivially implemented with weak semaphores. Since this does not accurately model the intuitive properties of weak semaphores discussed in Chapter 1, we reject this version of the finite delay property.

2.5 Solutions to the Starvation-Free Mutual Exclusion Problem

Our definition of starvation-free mutual exclusion will be complete once we have formalized the properties "desirable" for a solution to the mutual exclusion problem. The most important such property is the *mutual exclusion property.*

> **Definition 2.9** - A mutual exclusion system has the *mutual exclusion property* if there is no reachable state q, such that more than one process is in the critical region in state q.

A solution to the mutual exclusion problem would be of no use if it allowed the possibility of *deadlock*, which may be defined as follows:

> **Definition 2.10** - Process i in a mutual exclusion system is *deadlocked* in a state q, if process i is in the trying or leaving region in state q, and there is no finite execution sequence α, consisting only of transitions for processes in the trying region, critical region, or leaving region, such that process i is not in the trying or leaving region in state $nxt(q, \alpha)$. A mutual exclusion system is *deadlock-free* if no process is deadlocked in any reachable state.

In the preceding definition, to show that a process i in the trying or leaving region is not deadlocked in a given state, it is sufficient to construct a finite execution sequence from that state which moves process i into the critical region or noncritical region. The requirement that this execution sequence consist only of transitions for processes in the trying region, critical region, or leaving region is a consequence of our intuition that whether or not process i is deadlocked should not depend upon whether a particular process $j \neq i$ decides to leave the noncritical region at any given time.

Dijkstra [DIJKS66] requires of a solution to the mutual exclusion problem that, "If two processes are about to enter their critical regions, it must be impossible to devise for them such finite speeds, that the decision which one of the two is the first to enter its critical region is postponed to eternity." This statement implies that the system is free of a certain type of starvation. We will require that any solution to the mutual exclusion problem have a somewhat stronger version of this property, stated below. Although it seems reasonable to require that this property hold, it will not be needed for any of the results proved in this thesis.

> **Definition 2.11** - A mutual exclusion system is *free from indefinite postponement* if there is no valid infinite state sequence $q_0 q_1 \ldots$, with $q_0 = q_I$, such that for some k and all $j \geq k$ no process is in the critical region in state q_j.

The properties of mutual exclusion, freedom from deadlock, and freedom from indefinite postponement will be required for *all* solutions to the mutual exclusion problem. The additional property of *freedom from starvation* will be required for a solution to the *starvation-free* mutual exclusion problem. The difference between freedom from indefinite postponement and freedom from starvation is that the former applies to infinite execution sequences in which, after a certain point, *no* processes are in the critical region, whereas the latter is concerned with situations where a process remains forever in the trying or leaving region, while other processes execute *infinitely many* critical regions.

> **Definition 2.12** - A mutual exclusion system is *starvation-free* if there is no valid infinite state sequence $q_0 q_1 \ldots$, with $q_0 = q_I$, having the following properties:
>
> (1) There is a process i such that for all $k \geq 0$, there is a $m \geq k$ and

$n \geq k$ with process i in the critical region in state q_m, and process i not in the critical region in state q_n.

(2) There is a process i' and a $k' \geq 0$, such that for all $j \geq k'$, process i' is in the trying or leaving region in state q_j.

We may now define the term "solution to the (starvation-free) mutual exclusion problem".

Definition 2.13 - A *solution to the mutual exclusion problem for N processes* is a mutual exclusion system of N processes that has the mutual exclusion property, is deadlock-free, and is free from indefinite postponement. A *solution to the starvation-free mutual exclusion problem* is a solution to the mutual exclusion problem that is starvation-free.

2.6 Conclusion

In this chapter, parallel programs were introduced as a rather general model of parallel computation. A subclass of parallel programs, called "systems of processes" was defined, and we saw how the various types of semaphores could be modeled. Certain systems of processes, called "mutual exclusion systems", were identified, and it was seen that mutual exclusion systems model the important aspects of a system of processes competing for access to critical regions. Finally, we were able to use the model to precisely define a "solution to the (starvation-free) mutual exclusion problem". In the next chapter we will use these definitions in our investigation of the relationship betweeen semaphore primitives and starvation-free mutual exclusion.

3. Semaphore Primitives and Starvation-Free Mutual Exclusion

With the definitions of the previous chapter behind us, we may now proceed to the main task of this thesis: the investigation of the starvation properties of the various types of semaphores. The kind of question we will be attempting to answer is, "Can we solve the starvation-free mutual exclusion problem, using a particular type of semaphores?" Let us qualify this somewhat, by noting that it is trivial to produce a solution to the starvation-free mutual exclusion problem for $N < 2$ processes; therefore we assume throughout this chapter that $N \geq 2$. With this qualification in mind, we discover almost immediately that it is possible to solve the starvation-free mutual exclusion problem without using semaphores at all. The solution that illustrates this is due to Knuth [KNUTH66] and will be discussed in the next section. If we are to obtain any interesting information about the "power" of binary and general, weak and blocked-set semaphores with respect to their ability to solve the starvation-free mutual exclusion problem, then it appears that we must modify our question somewhat.

Knuth's semaphore-free solution to the starvation-free mutual exclusion problem has two properties that might be deemed "undesirable" if found in solutions using semaphores. We will term these two properties *busy-waiting* and *asymmetry.* Knuth's solution has busy-waiting because processes synchronize by looping and testing shared variables, rather than by becoming blocked. The solution is asymmetric because the program text executed by each process in the system depends explicitly on the process number. That is, the programs executed by the processes are not textually identical, as is usually the case in solutions that use semaphores. Note that the notion of symmetry intended here is quite strong. Precise definitions of busy-waiting and symmetry will be given later in this chapter. We will restrict our attention to *symmetric* mutual exclusion systems, and mutual exclusion systems with *no* busy-waiting. We will show that there are no semaphore-free symmetric solutions, and no semaphore-free solutions without busy-waiting. Furthermore, under these restrictions, differences in the "power" of the different kinds of semaphores will also become apparent.

The results proved in this chapter will be so-called "negative" results, which assert that certain classes of mutual exclusion systems contain no solutions to the starvation-free mutual exclusion problem. In Chapter 4, solutions will be exhibited in support of complementary "positive" results, which assert the existence of solutions in certain classes. The combination of negative and positive results will give us a somewhat clearer picture of the differences between the types of semaphores.

This chapter is organized as follows: After briefly examining Knuth's semaphore-free solution in the next section, we will proceed to the precise definition of symmetry and busy-waiting. A third property, *no memory* will also be defined. The assumption of "no memory" is natural in conjunction with the assumption of symmetry. Following the definitions will be a summary of the results of this chapter and the next. The remainder of this chapter will consist of the precise statement and proof of the negative results.

3.1 A Semaphore-Free Solution

Knuth's semaphore-free solution to the starvation-free mutual exclusion problem is displayed in Figure 3.1. It should be noted at this point that for purposes of understandability and compactness of presentation, any mutual exclusion system we discuss will be presented in an algorithmic notation similar to that of Figure 3.1. The reason we need the graphical notation of parallel programs at all is because our discussion will most often be at the level of single "atomic" state changes, and it is quite convenient to associate these state changes with the transitions of a parallel program. It should be a straightforward task for the reader to translate a mutual exclusion system presented in an algorithmic notation to the corresponding graphical form. When performing such a translation, care should be taken to be sure that each transition depends on or affects at most one global variable. Note also that although the program of Figure 3.1 uses a global array variable *control*, we have no provision for array variables in the parallel program model. The N-slot array *control* must therefore be replaced with N global variables, for example *control_1*, *control_2*, ... , *control_N*. The local variable j in Figure 3.1 may be eliminated by "unwinding" the loops.

For our purposes, the salient features of Knuth's solution are: (1) it has busy-waiting, since a process waits by looping in the trying region, and repeatedly examining the array *control*; and (2) it is asymmetric, since the program text executed by process i makes explicit reference to the process number i.

3.2 Restricted Classes of Mutual Exclusion Systems

We now turn to the precise definition of busy-waiting, symmetry, and no memory.

```
            global array control[1:N] initially 0;
            global k initially 1;

(Program Executed by Process i)

            local j
loop:       <noncritical region>
L0:         control[i] := 1;
L1:         for j := k step -1 until 1, N step -1 until 1 do
              begin if j = i then goto L2;
                if control[j] $\neq$ 0 then goto L1
              end;
L2:         control[i] := 2;
            for j := N step -1 until 1 do
              if $(j \neq i) \wedge (control[j] = 2)$ then goto L0;
            <critical region>
            k := if i = 1 then N else $i - 1$;
            control[i] := 0;
            goto loop;
```

Figure 3.1 - Knuth's Solution to the Starvation-Free Mutual Exclusion Problem

3.2.1 Busy-Waiting

As mentioned above, any solution to the mutual exclusion problem that has no semaphore operations must use busy-waiting to achieve synchronization. This is not difficult to prove, and we will do so presently, however we must first have a precise definition of "busy-waiting". The characteristic feature of a solution that uses busy-waiting is that no *a priori* bound may be placed on the amount of computation that a process may perform in the trying or leaving region. We attempt to capture this intution in the following:

> **Definition 3.1** - A mutual exclusion system *has busy waiting* if for any $M \geq 0$, there is a process *i*, numbers *m* and *n*, and an initial execution sequence $t_0 t_1 \ldots$, with corresponding state sequence $q_0 q_1 \ldots$, such that:
>
> (1) For all *j*, with $m \leq j \leq n$, process *i* is in the trying or leaving region in state q_j.
>
> (2) The number of $t_m, t_{m+1}, \ldots, t_n$ that are transitions for process *i* is at least *M*.

Let *NBW* be the class of all mutual exclusion systems that do not have busy-waiting. Since one of the reasons semaphores were introduced is to avoid the waste of processor time that is associated with busy-waiting, restricting our attention to solutions in *NBW* seems like a reasonable thing to do.

3.2.2 Symmetric Solutions

Another benefit of using semaphores to solve the mutual exclusion problem is that we obtain the ability to produce *symmetric* solutions. The notion of symmetry we are concerned with here is that of exact textual identity of program text when this text is presented in an algorithmic fashion as in Figure 3.1. It is possible to imagine weaker notions of symmetry, under which Knuth's program would be regarded as symmetric. An example of such a weaker notion would be to regard a program as symmetric if given the process number i, it is possible to compute the program text to be executed by process i.

Finding an appropriate definition of symmetry is somewhat complicated by the fact that although the motivation for the definition comes from comparing the algorithmic presentation of the program text for each process, we must make the definition in terms of the graphical notation of mutual exclusion systems. In the algorithmic notation, the queueing discipline of the semaphore operations is not made explicit, and therefore it is reasonable to define symmetric mutual exclusion systems to be those in which the code for all processes is textually identical. In the graphical notation however, the use of the process number within blocked-set semaphore operations is explicit, and therefore it is not possible for the graphs of two different processes, including the labels of the transitions, to be identical, as long as blocked-set semaphore operations are in use. We do not wish to consider a mutual exclusion system as asymmetric if the only difference between processes is the process number used for queueing purposes within blocked-set semaphore operations. Another reason why the definition cannot be made in this way is that whereas transitions in process i refer to one set of local variables, the transitions in process j refer to another. We do not wish to consider the system asymmetric if the only difference between processes i and j is that wherever process i refers to the local variable v^i, process j refers to the corresponding local variable v^j.

On the other hand, the definition of a symmetric mutual exclusion system must include the proviso that the local variables of one process have the same values in the initial state as the local variables of any other process. If this were not the case, then in the initial state, the local variables of each process could encode a process "identity". This identity could then be used by a process to select between different synchronization protocols. Thus, although the solution would be superficially symmetric, in essence it would be asymmetric. These ideas are incorporated into the following definitions:

> **Definition 3.2** - Processes i and j in a mutual exclusion system Γ are *structurally identical* if there is a graph isomorphism from $\mathfrak{X}_\Gamma^i$ to $\mathfrak{X}_\Gamma^j$ that preserves the critical place, the noncritical place, and semaphore operations.

We will indicate corresponding places and transitions in structurally identical processes by following a notational convention in which the process numbers are used as superscripts to the variables denoting the places and transitions. Thus, if p^i and t^i denote a place and transition in process i, then p^j and t^j denote the corresponding place and transition in the structurally identical process j.

It is convenient to define when two states "look the same" to some process in a mutual exclusion system. Informally, this occurs when the token for the process is at the same place in its graph in both states, and all global variables and variables local to the process have the same values in one state as they do in the other. Related to the concept of two states "looking alike" to a single process is the notion of a single state that "looks alike" to structurally identical processes in a system. The following definition captures both of these ideas.

> **Definition 3.3** - Suppose processes i and j in a mutual exclusion system Γ are structurally identical. If q and q' are states, then we say that *q looks to process i as q' looks to process j* (written $q \;{}_i{=}_j\; q'$) if:
>
> (1) For al! $p^i \in \mathcal{P}_\Gamma^i$, $q(p^i) = q'(p^j)$
>
> (2) For all $v \in \mathcal{G}_\Gamma$, $q(v) = q'(v)$
>
> (3) For all $v^i \in \mathcal{L}_\Gamma^i$, $q(v^i) = q'(v^j)$

If $q \; {}_{i}{\models}_{j} \; q$, then we say that state q *looks alike to processes i and j.* If $q \; {}_{i}{\models}_{i} \; q'$ then we say *q looks like q' to process i.*

Note that for any processes i, j, and k, and states q, q', and q'':

(1) $q \; {}_{i}{\models}_{i} \; q$

(2) $q \; {}_{i}{\models}_{j} \; q'$ iff $q' \; {}_{j}{\models}_{i} \; q$

(3) If $q \; {}_{i}{\models}_{j} \; q'$ and $q' \; {}_{j}{\models}_{k} \; q''$ then $q \; {}_{i}{\models}_{k} \; q''$

Two structurally identical processes i and j will be said to be *similar* if each transition t^i in process i has the "same effect" as the corresponding transition t^j in process j. If t^i and t^j are both transitions in a semaphore operation, then "same effect" is defined by condition (1) of Definition 3.4 below. Condition (1) states that the only difference between the labels of t^i and t^j is that wherever t^i refers to the process number i, transition t^j refers to the process number j. Condition (2) defines "same effect" if t^i and t^j are not part of semaphore operations, and states that the two transitions must have similar effects when fired from states that look the same. Note that similarity of processes is an equivalence relation.

Definition 3.4 - Two structurally identical processes i and j in a mutual exclusion system Γ are *similar* if for each pair $t^i \in \mathfrak{T}_\Gamma^i$ and $t^j \in \mathfrak{T}_\Gamma^j$ of corresponding transitions, either:

(1) Both t^i and t^j are part of semaphore operations, and there exist functions B: $\mathbb{N} \times \mathfrak{Q}_\Gamma$ ---> {**true, false**} and F: $\mathbb{N} \times \mathfrak{Q}_\Gamma$ ---> ($\mathfrak{V}_\Gamma$ ---> VAL) such that for all states q:

(a) $B_{t^i}(q) = B(i, q)$

(b) $B_{t^j}(q) = B(j, q)$

(c) $F_{t^i}(q) = F(i, q)$

(d) $F_{t^j}(q) = F(j, q)$

or (2) Neither t^i nor t^j is part of a semaphore operation, and for all states q and q' such that $q \,{}_i{\models}_j\, q'$:

(a) $B_{t^i}(q) = B_{t^j}(q')$

(b) $(F_{t^i}(q))(v) = (F_{t^j}(q'))(v)$ for all $v \in \mathcal{G}_\Gamma$

(c) $(F_{t^i}(q))(v^i) = (F_{t^j}(q'))(v^j)$ for all $v^i \in \mathcal{L}^i_\Gamma$

The following lemma is the basic property of similar processes, upon which we will base most of the results of this chapter.

> **Lemma 3.5** - Suppose processes i and j are similar processes in a mutual exclusion system Γ. Let t^i be a transition for process i, not part of a semaphore operation, and suppose that q and q' are states such that $q \,{}_i{\models}_j\, q'$. If t^i is enabled in state q then t^j is enabled in state q', and $nxt(q, t^i) \,{}_i{\models}_j\, nxt(q', t^j)$.

Proof - Suppose t^i is enabled in state q. This means that the token for process i is on the input place for transition t^i in state q. By the fact that $q \,{}_i{\models}_j\, q'$, and since processes i and j are similar, and hence have isomorphic graphs, we know that the token for process j must be on the input place for transition t^j in state q'. By Definition 3.4 we know that $B_{t^i}(q) = B_{t^j}(q')$, and hence transition t^j is enabled in state q'. Using Definition 2.2 and the fact that processes i and j have isomorphic graphs we have that

(1) $(nxt(q, t^i))(p^i) = (nxt(q', t^j))(p^j)$ for all $p^i \in \mathcal{P}^i_\Gamma$

In addition, Definition 3.4 implies that

(2) $(nxt(q, t^i))(v) = (nxt(q', t^j))(v)$ for all $v \in \mathcal{G}_\Gamma$.

(3) $(nxt(q, t^i)(v^i) = (nxt(q', t^j))(v^j)$ for all $v^i \in \mathcal{L}^i_\Gamma$.

Hence $nxt(q, t^i) \,{}_i{\models}_j\, nxt(q', t^j)$. ▓

> **Definition 3.6 -** A mutual exclusion system is *symmetric* if for every two processes *i* and *j*:
>
> (1) Processes *i* and *j* are similar
>
> (2) $q_I \; {}_i{=}_j \; q_I$

We will denote the class of all symmetric mutual exclusion systems by *SYM*.

3.2.3 No Memory

We will see in Chapter 4 (Example 4.3) that restricting our attention to *NBW* ∩ *SYM* gives us no more resolution of the differences between the types of semaphores than if we examine the larger class *NBW*. The reason for this, which is the central idea behind Example 4.3, is that any asymmetric solution to the starvation-free mutual exclusion problem in *NBW* may be used to construct a symmetric solution that is also in *NBW*. This construction requires the introduction of an additional semaphore variable, and an additional local variable in each process. Processes use the additional local variable to "remember" information about their synchronization history while they are in the noncritical region. We will say that a mutual exclusion that does *not* use local variables in this way has *no memory* and use *NM* to denote the class of all such systems. The precise definition of "no memory" may be given as follows:

> **Definition 3.7 -** A mutual exclusion system Γ has *no memory* if for each process *i*, local variable v for process *i*, and reachable state q such that process *i* is in the noncritical region in state q, $q(v) = q_I(v)$.

That is to say, whenever a process is in the noncritical region, its local variables have the same values as they do in the initial state q_I. It is obvious that this implies that the local variables cannot be used to "remember" information about past synchronization history.

Although there is no apparent reason why having no memory is a property of particularly "good" solutions to the mutual exclusion problem, it is pointed out in [BURNS79] that practically all solutions to the mutual exclusion problem in the literature have this property. We will see later on that within the class *SYM* ∩ *NM*, weak binary semaphores are strictly "weaker" than blocked-set binary semaphores, when their ability to implement starvation-free mutual exclusion is compared.

Note that it is not interesting to impose the requirement of no memory unless we also require symmetric solutions. This is because, in the absence of the assumption of symmetric solutions, global variables may be used to "simulate" the effect of local variables, by assigning each process some "private" global variables, which it alone can access. State information local to each process may then be stored in these variables. The requirement of symmetric solutions prohibits private global variables, since each process in a symmetric solution must access global variables in the same way as any other process.

3.3 Summary of Results

The results of this thesis are summarized below:

(1) If no restrictions are placed on the set of mutual exclusion systems allowed as solutions, then the starvation-free mutual exclusion problem can be solved without semaphores (Section 3.1).

(2) There are no semaphore-free solutions to the mutual exclusion problem in either *SYM* or *NBW* (Theorems 3.8 and 3.12). That is, requiring solutions either to be symmetric or to have no busy-waiting is sufficient to eliminate semaphore-free solutions.

(3) There exists a solution in *NBW* to the starvation-free mutual exclusion problem using either weak binary or weak general semaphores (Example 4.2). Although this solution is asymmetric, a simple transformation yields a solution in *NBW* ∩ *SYM* (Example 4.3). This solution, however, is *not* in

NM. Thus, although requiring solutions to be symmetric and have no busy-waiting eliminates the possibility of semaphore-free solutions, it does not rule out solutions that use weak semaphores.

(4) There exists a solution to the starvation-free mutual exclusion problem in $NBW \cap SYM \cap NM$, using either blocked-set binary or blocked-set general semaphores (Example 4.1). Thus the requirements of symmetry and no busy-waiting do not rule out solutions that use blocked-set semaphores either.

(5) There are no solutions to the starvation-free mutual exclusion problem in $SYM \cap NM$, which use weak binary semaphores (Theorem 3.14). In conjunction with (3), this result shows that although either weak binary or weak general semaphores can be used in a solution to the starvation-free mutual exclusion problem that is symmetric and has no busy-waiting, it is necessary to use "memory" to accomplish this. Together with (4), this result shows that weak binary semaphores are strictly "weaker" than either blocked-set binary or blocked-set general semaphores.

(6) There are no solutions to the starvation-free mutual exclusion problem in $NBW \cap SYM \cap NM$, that use weak general semaphores (Theorem 3.17). However, there is a solution for two processes in the larger class $SYM \cap NM$ (Example 4.4). Although it is unknown whether a solution exists in this class for more than two processes, any such solution must make use of local variables (Theorem 3.21). Thus, although weak general semaphores are slightly more "powerful" than weak binary semaphores, they are still strictly "weaker" than either type of blocked-set semaphores.

Figure 3.2 summarizes the results concerning the existence, in various restricted classes of mutual exclusion systems, of solutions to the starvation-free mutual exclusion problem using each of the four types of semaphores. A "Y" entry in the table indicates the existence of a solution, an "N" indicates that no solution exists. The "2" indicates that a solution for two processes is known to exist, although whether a solution exists for more than two processes is an open question. Four of the eight possible combinations of *NBW*, *SYM*, and *NM* are omitted from Figure 3.2. The reasons are the following: The unrestricted case is covered by (1) above; *NM* and $NBW \cap NM$ are eliminated by the observation in the previous section that requiring no memory is useless unless symmetric solutions are also required; and finally, it was previously observed that any solution in *NBW* may be transformed into a symmetric solution, thus eliminating case $NBW \cap SYM$.

Let us now proceed to the statement and proof of our results.

3.4 Every Semaphore-Free Solution Has Busy-Waiting

If $\alpha = t_0 t_1$ ... is an execution sequence for a parallel program, then let $seq(\alpha, i, j)$ denote the subsequence $t_i t_{i+1} \ldots t_{j-1}$ of α, if $i > j$, or Λ, if $i = j$. If α is infinite, then let $seq(\alpha, i, \infty)$ denote the infinite suffix $t_i t_{i+1} \ldots$, of α.

> **Theorem 3.8** - Every semaphore-free solution to the mutual exclusion problem has busy-waiting.

Proof - Let Γ be a semaphore-free solution to the mutual exclusion problem. By Definition 2.7, all processes are in the noncritical region in the initial state for Γ. Since Γ has no semaphores, every process has exactly one transition enabled in any reachable state. There is therefore a unique infinite initial execution sequence α for process 1, and it is easily seen that α is valid. (An execution sequence for process 1 is defined as an execution sequence consisting solely of transitions for process 1.) Because Γ is assumed deadlock-free, it follows that α contains a finite prefix $\alpha_m = seq(\alpha, 0, m)$, such that process 1 is in the critical region in state $nxt(q_I, \alpha_m)$.

	NBW	-	-	*NBW*
	-	*SYM*	*SYM*	*SYM*
	-	-	*NM*	*NM*
Weak Binary	Y	Y	N	N
Weak General	Y	Y	2	N
Blocked-set Binary	Y	Y	Y	Y
Blocked-set General	Y	Y	Y	Y

Figure 3.2 - Relative 'Power' of the Various Semaphore Primitives

Now, corresponding to each $n \geq 0$ is a unique finite execution sequence β_n of length n, such that β_n is an execution sequence for process 2 from $nxt(q_I, \alpha_m)$. Hence for any n, the sequence $\alpha_m \beta_n$ is a valid initial execution sequence for Γ. By the structure of the graph for process 2, and by the assumption that Γ has the mutual exclusion property, process 2 must be in the trying region in state $nxt(q_I, \alpha_m seq(\beta_n, 0, j))$ for all j with $0 \leq j \leq n$. Since n may be chosen arbitrarily large, Γ has busy-waiting. ▨

3.5 There Are No Semaphore-Free Symmetric Solutions

It was mentioned above that any solution to the mutual exclusion problem in *SYM* must use semaphores. The proof of this statement (Theorem 3.12) below depends in an essential fashion on an important property of symmetric mutual exclusion systems, which states that under certain conditions, two similar processes may execute in "lock-step" fashion, alternating the firing of corresponding transitions in their respective graphs. Each of the two processes executing in this way is "unaware" of the presence of the other. In fact, in some situations it is possible for two processes to execute in lock-step all the way from their noncritical regions to their critical regions. This hints that the idea of lock-step execution might be used as a technique for showing that a mutual exclusion system does not have the mutual exclusion property.

In this section, we will formalize the notion of lock-step execution of two processes, and will then apply it in the proof of Theorem 3.12. Before we can do this, however, we must prove a slightly generalized version (Lemma 3.10) of Lemma 3.5.

Definition 3.9 - Let i and j be structurally identical processes in a mutual exclusion system Γ, and let q and q' be states. Let $\mathcal{V} \subset \mathcal{G}_\Gamma \cup \mathcal{L}^i_\Gamma$. We say that $q \;{}_i{=}_j\; q'$ *except possibly for variables in* $\mathcal{V}$ if:

(1) $q(p^j) = q'(p^j)$ for all $p^j \in \mathcal{P}_\Gamma^i$

(2) $q(v) = q'(v)$ for all $v \in \mathcal{G}_\Gamma - \mathcal{V}$

(3) $q(v^j) = q'(v^j)$ for all $v^j \in \mathcal{L}_\Gamma^i - \mathcal{V}$.

Lemma 3.10 - Let i and j be similar processes in a mutual exclusion system Γ. Let t^i be a transition for process i, not part of a semaphore operation, and let q and q' be states such that $q \;{}_{i}{=}_{j}\; q'$ except possibly for variables not depended on by t^i. If t^i is enabled in state q, then t^j is enabled in state q', and $nxt(q, t^i) \;{}_{i}{=}_{j}\; nxt(q', t^j)$ except possibly for variables not affected by t^i.

Proof - Suppose t^i is enabled in state q. By Definition 3.4, transitions t^i and t^j affect and depend on the same global variables. In addition, a local variable $v^i \in \mathcal{L}_\Gamma^i$ is affected (depended on) by t^i if and only if v^j is affected (depended on) by t^j. Let $q_0, q_1, \dots, q_n$ be a sequence of states, with $q_0 = q'$ and $q_n \;{}_{j}{=}_{i}\; q$, such that for $0 \leq i < n$, state q_i is identical to q_{i+1} except for the value of a single variable not depended on by t^j. Such a sequence can be constructed since the set of variables not depended on by t^j is finite. By Lemma 3.5, t^j is enabled in state q_n, and $nxt(q, t^i) \;{}_{i}{=}_{j}\; nxt(q_n, t^j)$. Now, by Definition 2.4, for $0 \leq i < n$:

(1) $B_{t^j}(q_i) = B_{t^j}(q_{i+1})$; and

(2) $(F_{t^j}(q_i))(v) = (F_{t^j}(q_{i+1}))(v)$ for all variables v affected by t^j.

Hence $B_{t^j}(q') = B_{t^j}(q_n) = \textbf{true}$, and for all variables v affected by t^j, $(F_{t^j}(q'))(v) = (F_{t^j}(q_n))(v) = (nxt(q_n, t^j))(v)$. Therefore, t^j is enabled in state q', and

(3) $(nxt(q', t^j))(v) = (nxt(q_n, t^j))(v) = (nxt(q, t^i))(v)$ for all $v \in \mathcal{G}_\Gamma$ affected by t^i; and

(4) $(nxt(q', t^j))(v^j) = (nxt(q_n, t^j))(v^j) = (nxt(q, t^i))(v^i)$ for all $v^i \in \mathcal{L}_\Gamma^i$ affected by t^i.

Hence $nxt(q, t^i) \ {}_i{=}_j \ nxt(q', t^j)$ except possibly for variables not affected by t^i. ▪

If processes *1* and *2* in a mutual exclusion system are similar, and if $\alpha = t^1_0 t^1_1 \dots$ is an execution sequence for process 1, then let $alt(\alpha, i, j)$ denote the sequence $t^1_i t^2_i t^1_{i+1} t^2_{i+1} \dots t^1_{j-1} t^2_{j-1}$, if $j > i$, or Λ, if $j = i$. The following lemma formalizes the notion of lock-step execution. Although it requires that the sequence of transitions to be executed in lock-step include no transitions that are part of semaphore operations, this restriction will be eased in Lemma 3.15.

> **Lemma 3.11** - Let Γ be a mutual exclusion system of $N \geq 2$ processes, and suppose processes 1 and 2 in Γ are similar. Suppose further that $\alpha = t^1_0 t^1_1 \dots t^1_{m-1}$ is an execution sequence for process 1 from a state q_0, with corresponding state sequence $q_0 q_1 \dots q_m$. If $q_0 \ {}_1{=}_2 \ q_0$, then the execution sequence $\beta = alt(\alpha, 0, m)$ is also an execution sequence from q_0, provided that α contains no transitions that are part of semaphore operations. In addition, if $q'_m = nxt(q_0, \beta)$, then $q'_m \ {}_1{=}_2 \ q'_m$, $q'_m \ {}_1{=}_1 \ q_m$, and $q'_m(s) = q_m(s)$ for all semaphore variables s.

Proof - Suppose α contains no transitions that are part of semaphore operations. The proof is by induction on the length of α.

Base: If $\alpha = \Lambda$ then the Lemma holds trivially.

Induction Step: Suppose the Lemma holds for all sequences of length less than m, for some $m > 0$, and suppose α is of length m. Then application of the inductive hypothesis to the prefix $seq(\alpha, 0, m\text{-}1)$ of α shows that $alt(\alpha, 0, m\text{-}1)$ is an execution sequence from q_0. If $q'_{m-1} = nxt(q_0, alt(\alpha, 0, m\text{-}1))$, then $q'_{m-1} \ {}_1{=}_2 \ q'_{m-1}$, $q'_{m-1} \ {}_1{=}_1 \ q_{m-1}$, and $q'_{m-1}(s) = q_{m-1}(s)$ for all semaphore variables s.

To complete the proof, we will show that $alt(\alpha, m\text{-}1, m) = t^1_{m\text{-}1}t^2_{m\text{-}1}$ is an execution sequence from $q'_{m\text{-}1}$, and if $q'_m = nxt(q'_{m\text{-}1}, t^1_m t^2_m)$ then $q'_m\ {}_1{=}_2\ q'_m$ and $q'_m\ {}_1{=}_1\ q_m$. Note that since $q'_{m\text{-}1}(s) = q_{m\text{-}1}(s)$ for all semaphore variables s, and $t^1_{m\text{-}1}$ and $t^2_{m\text{-}1}$ do not affect any semaphore variables by hypothesis, it must be the case that $q'_m(s) = q_m(s)$ for all semaphore variables s.

Since $q'_{m\text{-}1}\ {}_1{=}_1\ q_{m\text{-}1}$, by Lemma 3.5 $t^1_{m\text{-}1}$ is enabled in $q'_{m\text{-}1}$, and if $q''_{m\text{-}1} = nxt(q'_{m\text{-}1}, t^1_{m\text{-}1})$, then $q''_{m\text{-}1}\ {}_1{=}_1\ q_m$. The remainder of the proof is split into two cases, depending upon whether $t^1_{m\text{-}1}$ affects a global variable or not.

Case 1: If $t^1_{m\text{-}1}$ affects no global variables, then $q''_{m\text{-}1}\ {}_2{=}_1\ q'_{m\text{-}1}$. Hence, by Lemma 3.5, $t^2_{m\text{-}1}$ is enabled in state $q''_{m\text{-}1}$, and if $q'_m = nxt(q''_{m\text{-}1}, t^2_{m\text{-}1})$, then $q'_m\ {}_2{=}_1\ q''_{m\text{-}1}$. Furthermore, since $t^1_{m\text{-}1}$ affects no global variables, neither does $t^2_{m\text{-}1}$, and hence $q'_m\ {}_1{=}_1\ q''_{m\text{-}1}\ {}_1{=}_1\ q_m$. Thus since $q'_m\ {}_2{=}_1\ q''_{m\text{-}1}$ and $q''_{m\text{-}1}\ {}_1{=}_1\ q'_m$, we have that $q'_m\ {}_2{=}_1\ q'_m$.

Case 2: If $t^1_{m\text{-}1}$ affects a global variable g, then neither $t^1_{m\text{-}1}$ nor $t^2_{m\text{-}1}$ can depend on any global variables. Hence $q''_{m\text{-}1}\ {}_2{=}_1\ q'_{m\text{-}1}$ except possibly for variables not depended on by $t^1_{m\text{-}1}$. But then by Lemma 3.10, $t^2_{m\text{-}1}$ is enabled in state $q''_{m\text{-}1}$, and if $q'_m = nxt(q''_{m\text{-}1}, t^2_{m\text{-}1})$, then $q'_m\ {}_2{=}_1\ q''_{m\text{-}1}$, except possibly for the values of variables not affected by $t^1_{m\text{-}1}$. This means that $q'_m(g) = q''_{m\text{-}1}(g)$, and since g is the only global variable affected by $t^2_{m\text{-}1}$, we know that $q'_m\ {}_1{=}_1\ q''_{m\text{-}1}\ {}_1{=}_1\ q_m$. That is, even though $t^2_{m\text{-}1}$ affects g, when fired from state $q''_{m\text{-}1}$ it assigns the same value to g as transition $t^1_{m\text{-}1}$ did when fired from state $q'_{m\text{-}1}$. Also, if $v^1 \in \mathcal{L}^1_{\mathrm{T}}$ is not affected by $t^1_{m\text{-}1}$, then $q'_m(v^1) = q''_{m\text{-}1}(v^1) = q'_{m\text{-}1}(v^1) = q'_{m\text{-}1}(v^2) = q''_{m\text{-}1}(v^2) = q'_m(v^2)$. Hence $q'_m\ {}_2{=}_1\ q''_{m\text{-}1}$, and therefore $q'_m\ {}_2{=}_1\ q'_m$. ■

Theorem 3.12 - Every solution to the mutual exclusion problem in *SYM* uses semaphores.

Proof - Suppose Γ is a solution to the mutual exclusion problem that is in *SYM* but uses no semaphores. As in the proof of Theorem 3.8, there is an initial execution sequence $\alpha_m = t_0^1 t_0^1 \ldots t_{m-1}^1$ for process 1, such that process 1 is in the critical region in state $q_m = nxt(q_1, \alpha_m)$. By Definition 3.6, $q_1 \ {}_1{=}_2 \ q_1$. Application of Lemma 3.11 shows that $\beta = alt(\alpha, 0, m)$ is an initial execution sequence for Γ, and that if $q'_m = nxt(q_1, \beta)$, then $q'_m \ {}_2{=}_1 \ q_m$ and $q'_m \ {}_1{=}_1 \ q_m$. But this means that processes 1 and 2 are both in the critical region in state q'_m, a contradiction with the assumption that Γ has the mutual exclusion property. ▩

3.6 Symmetric Weak Binary Semaphore Solutions With No Memory

In this section we will show that there are no symmetric solutions to the starvation-free mutual exclusion problem that use weak binary semaphores, and have the "no memory" property. Before proving this statement (Theorem 3.14), we must first introduce Lemma 3.13, which gives conditions relating two states q_0 and q'_0, sufficient to ensure that an execution sequence from q_0 is also an execution sequence from q'_0. Lemma 3.13 applies only to mutual exclusion systems with weak binary semaphores, however later on we will state a version (Lemma 3.16) applicable to systems with weak general semaphores.

When we discuss systems with weak semaphores, note that we may, without confusion, make the statements "transition t is a **P** operation on a semaphore variable s", or "transition t is a **V**(s) operation". This is because each weak semaphore operation contains only a single transition. Also note that in a system with weak semaphores, there are no synchronization variables other than semaphore variables.

Lemma 3.13 - Let Γ be a mutual exclusion system with weak binary semaphores. Suppose α is an execution sequence for process 1 from a state q_0. If q'_0 is a state such that $q'_0 \;{}_1{=}_1\; q_0$ and if $q'_0(s) \geq q_0(s)$ for all $s \in \mathfrak{S}_\Gamma$, then α is an execution sequence from q'_0 as well.

Proof - Note that it suffices to prove the Lemma for finite sequences α, since if α is infinite, then application of the finite case shows that any finite prefix of α is an execution sequence from q'_0, and hence α itself is an execution sequence from q'_0.

The proof for the finite case is by induction on the length of α. However, it is convenient to prove a somewhat stronger result, namely in addition to proving that α is an execution sequence from q'_0, we will show that if $q_m = nxt(q_0, \alpha)$ and $q'_m = nxt(q'_0, \alpha)$, then $q_m \;{}_1{=}_1\; q'_m$ and $q'_m(s) \geq q_m(s)$ for all $s \in \mathfrak{S}_\Gamma$.

Base: If $\alpha = \Lambda$, then the result holds trivially.

Induction Step: Suppose the result holds for all sequences of length less than m, for some $m > 0$, and suppose $\alpha = t_0 t_1 \ldots t_{m-1}$ is of length m. Then application of the inductive hypothesis to the prefix $seq(\alpha, 0, m\text{-}1)$ of α shows that $seq(\alpha, 0, m\text{-}1)$ is an execution sequence from q'_0, and if $q_{m-1} = nxt(q_0, seq(\alpha, 0, m\text{-}1))$ and $q'_{m-1} = nxt(q'_0, seq(\alpha, 0, m\text{-}1))$, then $q_{m-1} \;{}_1{=}_1\; q'_{m-1}$ and $q'_{m-1}(s) \geq q_{m-1}(s)$ for all $s \in \mathfrak{S}_\Gamma$. It remains to be shown that t_{m-1} is enabled in state q'_{m-1}, and if $q_m = nxt(q_{m-1}, t_{m-1})$ and $q'_m = nxt(q'_{m-1}, t_{m-1})$, then $q_m \;{}_1{=}_1\; q'_m$ and $q'_m(s) \geq q_m(s)$ for all $s \in \mathfrak{S}_\Gamma$. The proof of this is split into three cases, depending upon whether t_{m-1} is a **P** operation, a **V** operation, or neither.

Case 1: Transition t_{m-1} is neither a **P** operation nor a *V* operation. Then Lemma 3.5 shows that t_{m-1} is enabled in state q'_{m-1} and that $q_m \;{}_1{=}_1\; q'_m$. Since t_{m-1} affects no semaphore variables, we know that for all $s \in \mathfrak{S}_\Gamma$, $q'_m(s) = q'_{m-1}(s) \geq$

$q_{m-1}(s) = q_m(s)$.

Case 2: Transition t_{m-1} is a **V** operation on a semaphore variable $\bar{s}$. Then t_{m-1} is enabled in state q'_{m-1} because t_{m-1} is enabled in state q_{m-1}, $q_{m-1} \;{}_{1}{=}_{1}\; q'_{m-1}$, and the enabling predicate of a weak **V** operation is always true. Now t_{m-1} affects only $\bar{s}$ and hence $q_m \;{}_{1}{=}_{1}\; q'_m$. Also $q'_m(\bar{s}) = 1 = q_m(\bar{s})$ and for all $s \in \mathfrak{S}_{\Gamma}$ with $s \neq \bar{s}$ we know that $q'_m(s) = q'_{m-1}(s) \geq q_{m-1}(s) = q_m(s)$.

Case 3: Transition t_{m-1} is a **P** operation on a semaphore variable $\bar{s}$. Then t_{m-1} is enabled in state q'_{m-1} if $q'_{m-1}(\bar{s}) > 0$. But since t_{m-1} is enabled in state q_{m-1}, we know that $q_{m-1}(\bar{s}) > 0$. Since $q'_{m-1}(\bar{s}) \geq q_{m-1}(\bar{s})$, we know that $q'_{m-1}(\bar{s}) > 0$ as well. Since t_{m-1} affects only $\bar{s}$ we have that $q_m \;{}_{1}{=}_{1}\; q'_m$ and that for all $s \in \mathfrak{S}_{\Gamma}$ with $s \neq \bar{s}$, $q'_m(s) \geq q_m(s)$ as in Case 2. In addition $q'_m(\bar{s}) = q'_{m-1}(\bar{s}) - 1 \geq q_{m-1}(\bar{s}) - 1 = q_m(\bar{s})$. ■

> **Theorem 3.14** - There is no solution to the starvation-free mutual exclusion problem in *SYM* ∩ *NM* that has weak binary semaphores.

Proof - Suppose Γ is a solution to the starvation-free mutual exclusion problem that is in *SYM* ∩ *NM*. Suppose further that Γ has weak binary semaphores. We will construct an execution sequence for Γ that starves process 2. Since this contradicts the assumption that Γ is starvation-free, we conclude that Γ cannot exist.

Now, the assumption that Γ is deadlock-free may be used to show that there is a unique infinite initial execution sequence α for process 1, in which process 1 enters and exits its critical region infinitely often. Let $q_0 q_1 \ldots$ be the state sequence corresponding to α. Define the indices $ncr_1, ncr_2, \ldots$, and $cr_1, cr_2, \ldots$ as follows:

(1) Let $ncr_1 = 0$, and let cr_1 be the least $j \geq 0$ such that process 1 is in the critical region in state q_j

(2) For each $i > 1$ let ncr_1 be the least $j > cr_{i-1}$ such that process 1 is in the noncritical region in state $q_j.$ Similarly, let cr_i be the least $j > ncr_i$ such that process 1 is in the critical region in state $q_j.$

Let us first show that for each $i > 0$, $seq(\alpha, ncr_i, cr_i)$ contains at least one **P** operation. To see this, suppose there were no **P** operations in $seq(\alpha, ncr_i, cr_i)$ for some i. Because Γ is in $SYM \cap NM$, we know that processes 1 and 2 are similar, and that $q_{ncr_i}\ {}_1{=}_2\ q_{ncr_i}.$ Application of Lemma 3.11 shows that $alt(\alpha, ncr_i, cr_i)$ is also an execution sequence from q_{ncr_i}, and that if $q'_{cr_i} = nxt(q_{ncr_i}, alt(\alpha, ncr_i, cr_i))$ then $q'_{cr_i}\ {}_1{=}_2\ q'_{cr_i}$ and $q'_{cr_i}\ {}_1{=}_1\ q_{cr_i}.$ Hence processes 1 and 2 are both in the critical region in the state q'_{cr_i}, a contradiction with the assumption that Γ has the mutual exclusion property.

Thus each $seq(\alpha, ncr_i, cr_i)$ must contain at least one **P** operation. Let the indices $p_1, p_2, \ldots$ be defined so that $t^1_{p_i}$ is the first **P** operation in $seq(\alpha, ncr_i, cr_i)$, and let s_i be the corresponding semaphore variable. Since $\mathbf{S}_\Gamma$ is finite, but there are infinitely many s_i, there must be one semaphore variable $\bar{s}$, such that $s_i = \bar{s}$ for infinitely many i. Let k be the least i for which $s_i = \bar{s}$. We will now show that $\beta = seq(\alpha, 0, ncr_k)alt(\alpha, ncr_k, p_k)seq(\alpha, p_k, \infty)$ is an initial execution sequence for Γ that starves process 2.

Obviously $seq(\alpha, 0, ncr_k)$ is an initial execution sequence for Γ. Now $seq(\alpha, ncr_k, p_k)$ contains no **P** operations and $q_{ncr_k}\ {}_1{=}_2\ q_{ncr_k}.$ Application of Lemma 3.11 shows that $alt(\alpha, ncr_k, p_k)$ is an execution sequence from state $q_{ncr_k}.$ In addition if $q'_{p_k} = nxt(q_{ncr_k}, alt(\alpha, ncr_k, p_k))$, then $q'_{p_k}\ {}_1{=}_2\ q'_{p_k}$ and $q'_{p_k}\ {}_1{=}_1\ q_{p_k}.$ Thus both processes 1 and 2 are at the input place for a $\mathbf{P}(\bar{s})$ in state $q'_{p_k}.$ In addition, all semaphore variables have the same values in state q'_{p_k} as they do in state state $q_{p_k}.$ Lemma 3.13 therefore shows that $seq(\alpha, p_k, \infty)$ is an execution sequence from q'_{p_k} for Γ. Thus β is an infinite initial execution sequence for Γ.

To show that β starves process 2, note that process 1 enters and exits its critical region infinitely often in β, while process 2 remains forever in the trying region at the input place for a $\mathrm{P}(\bar{s})$. Since $seq(\alpha, p_k, \infty)$ contains infinitely many $\mathrm{P}(\bar{s})$ operations, and each $\mathrm{P}(\bar{s})$ sets $\bar{s}$ to zero, process 2 is disabled infinitely often in β. The sequence β is therefore a valid execution sequence in which process 2 starves. This contradicts the assumption that Γ is starvation-free, and we conclude that Γ cannot exist. ▨

3.7 Symmetric Weak General Semaphore Solutions With No Memory

Theorem 3.17 in this section shows that any weak general semaphore solution to the starvation-free mutual exclusion problem that is in $SYM \cap NM$ cannot be in *NBW*. Before we can prove this though, we need versions of Lemmas 3.11 and 3.13 that apply to execution sequences containing weak general semaphore operations.

Recall that Lemma 3.11 gave conditions under which two similar processes could execute a sequence of transitions in lock-step. However, Lemma 3.11 only applies if the sequence of transitions contains no transitions that are part of semaphore operations. This is too restrictive for systems with weak general semaphores, since even semaphore operations can be executed in lock-step, as long as the values of the semaphore variables are "great enough." Lemma 3.15 below makes this idea precise.

If α is a finite sequence of transitions for a mutual exclusion system with weak general semaphores, and if s is a semaphore variable, then define the ***index of* α *with respect to* s**, denoted $ind(s, \alpha)$, to be the number of $\mathrm{P}(s)$ transitions in α minus the number of $\mathrm{V}(s)$ transitions in α. The notion of the index of a sequence of transitions is useful for the following reason: If q is a state, α is an execution sequence from q, and s is a semaphore variable, then $(nxt(q, \alpha))(s) = q(s) - ind(s, \alpha)$.

Lemma 3.15 - Let Γ be a mutual exclusion system of $N \geq 2$ processes which has weak general semaphores, and suppose processes 1 and 2 in Γ are similar. Suppose further that $\alpha = t_0^1 t_1^1 \dots t_{m-1}^1$ is an execution sequence for process 1 from q_0, with corresponding state sequence $q_0 q_1 \dots q_m$. If $q_0 \;{}_1{=}_2\; q_0$, then the execution sequence $\beta = alt(\alpha, 0, m)$ is also an execution sequence from q_0, provided that for all $s \in \mathcal{S}_\Gamma$ and $0 \leq j \leq m$, $q_0(s) - 2 \cdot ind(s, seq(\alpha, 0, j)) \geq 0$. In addition, if $q'_m = nxt(q_0, \beta)$, then $q'_m \;{}_1{=}_2\; q'_m$, $q'_m \;{}_1{=}_1\; q_m$, and for all $s \in \mathcal{S}_\Gamma$, $q'_m(s) = q_0(s) - 2 \cdot ind(s, \alpha)$.

Proof - Suppose that for all $s \in \mathcal{S}_\Gamma$ and $0 \leq j \leq m$, $q_0(s) - 2 \cdot ind(s, seq(\alpha, 0, j)) \geq 0$. The proof is by induction on the length of α.

Base: If $\alpha = \Lambda$ then the Lemma holds trivially.

Induction Step: Suppose the Lemma holds for all sequences of length less than m for some $m > 0$, and suppose α is of length m. Application of the inductive hypothesis to the prefix $seq(\alpha, 0, m\text{-}1)$ of α shows that $alt(\alpha, 0, m\text{-}1)$ is an execution sequence from q_0. If $q'_{m-1} = nxt(q_0, alt(\alpha, 0, m\text{-}1))$, then $q'_{m-1} \;{}_1{=}_2\; q'_{m-1}$ and $q'_{m-1} \;{}_1{=}_1\; q_{m-1}$. In addition, for all $s \in \mathcal{S}_\Gamma$, $q'_{m-1}(s) = q_0(s) - 2 \cdot ind(s, seq(\alpha, 0, m\text{-}1))$.

To complete the proof, we will show that $alt(\alpha, m\text{-}1, m) = t_{m-1}^1 t_{m-1}^2$ is an execution sequence from q'_{m-1}, and if $q'_m = nxt(q'_{m-1}, t_m^1 t_m^2)$ then $q'_m \;{}_1{=}_2\; q'_m$ and $q'_m \;{}_1{=}_1\; q_m$. In addition, we will show that $q'_m(s) = q_0(s) - 2 \cdot ind(s, \alpha)$ for all $s \in \mathcal{S}_\Gamma$.

There are three cases, depending upon whether t_{m-1}^1 (and hence t_{m-1}^2) is a **P** operation, a **V** operation, or neither.

Case 1: Transition t_{m-1}^1 is not part of a semaphore operation. In this case, an application of Lemma 3.11 completes the proof.

Case 2: Transition t^1_{m-1} is a V operation on some semaphore variable $\bar{s}$. Since the enabling predicate of a V operation is always true, it is clear that $t^1_{m-1}t^2_{m-1}$ is an execution sequence from q'_{m-1}. Now, t^1_{m-1} and t^2_{m-1} affect only the semaphore variable $\bar{s}$. It therefore follows easily that $q'_m \;{}_1{=}_1\; q_m$ and $q'_m \;{}_1{=}_2\; q'_m$. The effect of executing $t^1_{m-1}t^2_{m-1}$ is simply to increment $\bar{s}$ twice. Hence for all $s \in \mathbb{S}_\Gamma$ with $s \neq \bar{s}$, $q'_m(s) = q'_{m-1}(s) = q_0(s) - 2{\cdot}ind(s, seq(\alpha, 0, m\text{-}1)) = q_0(s) - 2{\cdot}ind(s, \alpha)$. Also $q'_m(\bar{s}) = q'_{m-1}(\bar{s}) + 2 = q_0(\bar{s}) - 2{\cdot}ind(\bar{s}, seq(\alpha, 0, m\text{-}1)) + 2 = q_0(\bar{s}) - 2{\cdot}ind(\bar{s}, \alpha)$.

Case 3: Transition t^1_{m-1} is a P operation on some semaphore variable $\bar{s}$. Now, by assumption $q_0 - 2{\cdot}ind(\bar{s}, \alpha) \geq 0$. But since $q'_{m-1}(\bar{s}) = q_0 - 2{\cdot}ind(\bar{s}, seq(\alpha, 0, m\text{-}1)) = q_0 - 2{\cdot}ind(\bar{s}, \alpha) + 2$, we have that $q'_{m-1}(\bar{s}) \geq 2$. But this means that $t^1_{m-1}t^2_{m-1}$ is an execution sequence from q'_{m-1}. Since t^1_{m-1} and t^2_{m-1} affect only $\bar{s}$, we know that $q'_m \;{}_1{=}_1\; q_m$ and $q'_m \;{}_1{=}_2\; q'_m$ as in Case 2. Also as in Case 2, $q'_m(s) = q_0(s) - 2{\cdot}ind(s, \alpha)$ for all $s \in \mathbb{S}_\Gamma$ with $s \neq \bar{s}$. Finally, $q'_m(\bar{s}) = q'_{m-1}(\bar{s}) - 2 = q_0(\bar{s}) - 2{\cdot}ind(\bar{s}, seq(\alpha, 0, m\text{-}1)) - 2 = q_0(\bar{s}) - 2{\cdot}ind(\bar{s}, \alpha)$. ▩

Lemma 3.16 - Let Γ be a mutual exclusion system with weak general semaphores. Suppose α is an execution sequence for process 1 from a state q. If $q' \;{}_1{=}_1\; q$ and if $q'(s) - ind(s, seq(\alpha, 0, j)) \geq 0$ for all $s \in \mathbb{S}_\Gamma$ and $0 \leq j \leq length(\alpha)$, then α is an execution sequence from q' as well.

Proof - The proof, which is quite similar to the proof of Lemma 3.13, is left to the reader. ▩

Theorem 3.17 - Any solution to the starvation-free mutual exclusion problem that has weak general semaphores and is in $SYM \cap NM$, also has busy-waiting.

The construction in the proof of this theorem is somewhat more involved than those that have appeared so far, and it will be convenient first to separate out some reasonably independent parts of the proof as Lemma 3.18, Lemma 3.19, and Corollary 3.20.

For the remainder of this section, let Γ be a solution to the starvation-free mutual exclusion problem that has weak general semaphores and is in $SYM \cap NM$. Let $\alpha = t_0^1 t_1^1 \ldots$ be the (unique) infinite initial execution sequence for process 1, and let the indices $ncr_1, ncr_2, \ldots$, and $cr_1, cr_2, \ldots$ be constructed as in the proof of Theorem 3.14. Let $q_0 q_1 \ldots$ be the state sequence corresponding to the execution sequence α. If $s \in \mathbb{S}_\Gamma$, then let $\#_{\mathbf{P}(s)}(i, j)$ denote the number of $\mathbf{P}(s)$ operations in the subsequence $seq(\alpha, i, j)$ of α. Let $ind(s, i, j)$ abbreviate $ind(s, seq(\alpha, i, j))$. Define $val(s, i, j) = q_i(s) - 2 \cdot ind(s, i, j)$. Intuitively, $val(s, i, j)$ represents the value that the semaphore variable s would have if processes 1 and 2 were to execute the "lock-step" execution sequence $alt(\alpha, i, j)$ from state q_i.

We will now investigate some properties of the execution sequence α. The intuition behind Lemma 3.18 below is that if processes 1 and 2 begin executing in "lock-step" from state q_{ncr_i} for some $i \geq 1$, then there must be some $\mathbf{P}$ operation before both processes reach the critical region, where the processes are forced to "split up".

> **Lemma 3.18** - For each $i \geq 1$ there exists a number $split_i$, which is the least j, $ncr_i \leq j < cr_i$, such that t_j^1 is a $\mathbf{P}$ operation on some semaphore s with $val(s, ncr_i, j) \leq 2$. Moreover, if $\hat{s}_i$ denotes the semaphore variable on which $t^1_{split_i}$ operates, then $val(\hat{s}_i, ncr_i, split_i) = 1$.

Proof - If $val(s, ncr_i, j) \geq 2$ for each j with $ncr_i \leq j < cr_i$ and t_j^1 a $\mathbf{P}$ operation on some semaphore variable s, then we could apply Lemma 3.15 to show that

$alt(\alpha, ncr_i, cr_i)$ is an execution sequence from q_{ncr_i}. Since this means that processes 1 and 2 would both be in the critical region in state $nxt(q_{ncr_i}, alt(\alpha, ncr_i, cr_i))$, it must be the case that there is at least one j with $ncr_i \leq j < cr_i$ and t_j^1 a **P** operation on some semaphore s, such that $val(s, ncr_i, j) < 2$. Let $split_i$ be the least such j, and let $\hat{s}_i$ be the corresponding semaphore variable.

It remains to be shown that $val(\hat{s}_i, ncr_i, split_i) \neq 0$. Suppose the contrary. By the construction of $split_i$ in the preceding paragraph, we know that $val(s, ncr_i, j) \geq 0$ for all semaphore variables s and all j with $ncr_i \leq j \leq split_i$. We may therefore apply Lemma 3.15 to show that $alt(\alpha, ncr_i, split_i)$ is an execution sequence from q_{ncr_i}. Furthermore, if we let $q'_{split_i} = nxt(q_{ncr_i}, alt(\alpha, ncr_i, split_i))$, then $q'_{split_i}(\hat{s}_i) = 0$. But processes 1 and 2 are both at the input place for a $\mathbf{P}(\hat{s}_i)$ in state q'_{split_i}, and hence are deadlocked. This is a contradiction, and we conclude that $val(\hat{s}_i, ncr_i, split_i) = 1$ as asserted. ▓

Processes 1 and 2 can execute the sequence $alt(\alpha, ncr_i, split_i)$ from q_{ncr_i} before being forced to split up, but there is no guarantee that process 1 will be able to continue alone unhindered. We would like an answer to the question of how far it is "safe" for processes 1 and 2 to execute in lock-step from q_{ncr_i}, if process 1 is then to continue alone unhindered. The answer to this question depends upon how great the values of the semaphore variables are in state q_{ncr_i}, and is given by Corollary 3.20 below.

> **Lemma 3.19** - Let $i > 0$ and let c: $\mathbb{S}_{\mathrm{T}}$ ---> $\mathbb{N}$, with the property that $q_j(s) \geq c(s)$ for all $j \geq ncr_i$. Then there exists a number $safe_i$, which is the least j, $ncr_i \leq j \leq split_i$, such that transition t_j^1 is a **P** operation on some semaphore variable s, and $ind(s, ncr_i, j) = c(s)$.

Proof - It suffices to show the existence of one such j. We break the proof into three cases.

Case 1: $ind(\hat{s}_i, ncr_i, split_i) = c(\hat{s}_i)$. In this case, let $j = split_i$ and $s = \hat{s}_i$.

Case 2: $ind(\hat{s}_i, ncr_i, split_i) < c(\hat{s}_i)$. By the definition of $split_i$ we know that $val(\hat{s}_i, ncr_i, split_i) = q_{ncr_i}(\hat{s}_i) - 2 \cdot ind(\hat{s}_i, ncr_i, split_i) = 1$, and hence

$$(1)\ q_{ncr_i}(\hat{s}_i) - ind(\hat{s}_i, ncr_i, split_i) = ind(\hat{s}_i, ncr_i, split_i) + 1$$

Since $t^l_{split_i}$ is a $\mathbf{P}(\hat{s}_i)$, we know that $q_{split_i+1}(\hat{s}_i) = q_{split_i}(\hat{s}_i) - 1 \geq c(\hat{s}_i)$. Hence

$$\begin{aligned} c(\hat{s}_i) &\leq q_{split_i}(\hat{s}_i) - 1 \\ &= q_{ncr_i}(\hat{s}_i) - ind(\hat{s}_i, ncr_i, split_i) - 1 \\ &= ind(\hat{s}_i, ncr_i, split_i), \text{ by (1)} \\ &< c(\hat{s}_i), \text{ by the assumption defining Case 2} \end{aligned}$$

This is a contradiction, and we conclude that Case 2 is impossible.

Case 3: $ind(\hat{s}_i, ncr_i, split_i) > c(\hat{s}_i)$. Let $I(j)$ abbreviate $ind(\hat{s}_i, ncr_i, j)$ and let the predicate $Q(j)$ be true iff $I(j) \leq c(\hat{s}_i)$. Since $I(ncr_i) = 0$ we know that $Q(ncr_i)$ is true. Now, suppose $Q(j)$ is true for some j with $ncr_i \leq j \leq cr_i$. If t^l_j is not a $\mathbf{P}(\hat{s}_i)$, then $Q(j+1)$ is true. If t^l_j is a $\mathbf{P}(\hat{s}_i)$, then $Q(j+1)$ is false iff $I(j) = c(\hat{s}_i)$. Since $Q(split_i)$ is false and $Q(ncr_i)$ is true, this means that there must be a j with $ncr_i \leq j \leq split_i$, such that t^l_j is a $\mathbf{P}(\hat{s}_i)$ and $I(j) = ind(\hat{s}_i, ncr_i, j) = c(\hat{s}_i)$. This completes the proof. ▩

Corollary 3.20 - If $ncr_i \leq j \leq safe_i$, and if $q'_j = nxt(q_{ncr_i}, alt(\alpha, ncr_i, j))$ then $seq(\alpha, j, \infty)$ is an execution sequence from q'_j.

Proof - The Corollary is proved by an application of Lemma 3.16 after observing that for all $s \in \mathbb{S}_\Gamma$, $q'_j(s) \geq q_j(s) - c(s)$, and for all $k \geq j$, $q_j(s) - ind(s, j, k) \geq c(s)$. ▩

Proof of Theorem 3.17 - We will show that Γ has busy-waiting by showing the existence of a semaphore variable $\bar{s}$ such that for any $M \geq 0$ there is an $m > 0$ with $\#_{\mathbf{P}(\bar{s})}(ncr_m, cr_m) > M$. This will be accomplished as follows: We will show the existence of a sequence of functions c_0, c_1, ... mapping $\mathbb{S}_\Gamma$ to $\mathbb{N}$ such that:

(1) For all $i \geq 0$, $c_i < c_{i+1}$, where $c_i < c_{i+1}$ is defined to mean that for all semaphore variables s, $c_i(s) \leq c_{i+1}(s)$, and strict inequality holds for at least one s.

Since there are only a finite number of semaphore variables, property (1) implies that there must be a semaphore variable $\bar{s}$ such that $c_i(\bar{s})$ increases monotonically with increasing i. We will show in addition the existence of a sequence of natural numbers $k(0)$, $k(1)$, ... , with the properties:

(2) For all $i \geq 0$, for all $j \geq ncr_{k(i)}$, and for all semaphore variables s, $q_j(s) \geq c_i(s)$.

(3) For any n there is an $m \geq n$ such that $\#_{\mathbf{P}(\bar{s})}(ncr_{k(m)}, cr_{k(m)}) \geq c_m(\bar{s})$

Properties (2) and (3) show that Γ has busy-waiting, since given $M \geq 0$ we may choose n such that $c_n(\bar{s}) \geq M$, then select $m \geq n$ so that $\#_{\mathbf{P}(\bar{s})}(ncr_{k(m)}, cr_{k(m)}) \geq c_m(\bar{s})$.

We now turn to the inductive construction of the functions c_i and numbers $k(i)$.

Base: Define $k(0) = 1$, and let $c_0(s) = 0$ for all $s \in \mathbb{S}_\Gamma$. Obviously, for all $j \geq ncr_{k(0)}$ and all $s \in \mathbb{S}_\Gamma$, $q_j(s) \geq c_0(s)$.

Induction Step: Suppose for some $i \geq 0$ we have a function c_i and number $k(i)$ such that for all $j \geq ncr_{k(i)}$ and all $s \in \mathfrak{S}_\Gamma$, $q_j(s) \geq c_i(s)$.

Lemma 3.19 and Corollary 3.20 show the existence of $safe_{k(i)}$, such that $\beta = alt(\alpha, ncr_{k(i)}, safe_{k(i)})seq(\alpha, safe_{k(i)}, \infty)$ is an execution sequence from $q_{ncr_{k(i)}}$ for Γ. In β, process 1 executes infinitely many critical regions while process 2 remains in the trying region, at the input place for a **P** operation on some semaphore, call it $\tilde{s}_i$. If β were valid, then we would have a contradiction with the assumption that Γ is starvation-free. Hence β must not be valid. The only way for this to occur is if there exists $k(i+1) > k(i)$ such that $q'_j(\tilde{s}_i) > 0$ for all $j \geq ncr_{k(i+1)}$, where $q'_j = nxt(q_{ncr_{k(i)}}, alt(\alpha, ncr_{k(i)}, safe_{k(i)})seq(\alpha, safe_{k(i)}, j))$. But since $q'_j(\tilde{s}_i) > 0$ and $q'_j(\tilde{s}_i) = q_j(\tilde{s}_i) - c_i(\tilde{s}_i)$ we have that $q_j(\tilde{s}_i) \geq c_i(\tilde{s}_i) + 1$. Define $c_{i+1}(s) = c_i(s)$ for all $s \in \mathfrak{S}_\Gamma$ with $s \neq \tilde{s}_i$, and define $c_{i+1}(\tilde{s}_i) = c_i(\tilde{s}_i) + 1$. This completes the construction of c_{i+1} and $k(i+1)$ from c_i and $k(i)$.

It is easy to see that the c_i and $k(i)$ defined in this way have properties (1) and (2). To see that (3) holds as well, let $\bar{s}$ be the semaphore variable, whose existence was argued above, such that $c_i(\bar{s})$ increases monotonically with increasing i. It must be the case that for infinitely many i, $\bar{s}$ is the semaphore $\tilde{s}_i$. Since for each such i, $ind(\bar{s}, ncr_{k(i)}, safe_{k(i)}) = c_i(\bar{s})$, and since $\#_{\mathbf{P}(\bar{s})}(ncr_{k(i)}, safe_{k(i)}) \geq ind(\bar{s}, ncr_{k(i)}, safe_{k(i)})$, property (3) holds as well. ▨

3.8 More on Weak General Semaphore Solutions

We have seen that any solution to the starvation-free mutual exclusion problem that has weak general semaphores and is in $SYM \cap NM$, must also have busy-waiting. In Chapter 4, we will actually exhibit such a solution for the special case of two processes. Although it is unknown whether this solution extends to the case of more than two processes, we conclude this chapter with a result that shows that such a solution must make use of local variables.

Theorem 3.21 - Any solution to the starvation-free mutual exclusion problem that has weak general semaphores and is in $SYM \cap NM$, must contain local variables.

Proof - Suppose Γ is a solution to the starvation-free mutual exclusion problem, that has weak general semaphores and is in $SYM \cap NM$. Let us inherit all the notation of the previous section pertaining to the execution sequence α for Γ. Suppose, to obtain a contradiction, that Γ has no local variables. We will construct an execution sequence for Γ that starves process 2.

From among the numbers $k(0)$, $k(1)$, ... defined in the proof of Theorem 3.17, select numbers $m(0) < m(1) < \ldots$, such that $\#_{\mathbf{P}(\bar{s})}(ncr_{m(i)}, safe_{m(i)}) < \#_{\mathbf{P}(\bar{s})}(ncr_{m(i+1)}, safe_{m(i+1)})$ for all $i \geq 0$. Let $\alpha_i = seq(\alpha, ncr_{m(i)}, safe_{m(i)})$. Thus the $m(i)$ are chosen to make the number of $\mathbf{P}(\bar{s})$ operations in α_i a monotonically increasing function of i.

Let $\mathfrak{A}_{m(i)}$ be the set of all $t \in \mathfrak{T}'_{\Gamma}$ such that t appears in α_i and t is a $\mathbf{P}$ operation. Call the pair (t, t') a *link* in α_i if t and t' are both in $\mathfrak{A}_{m(i)}$ and $t\gamma t'$ is a subsequence of α_i such that γ contains no $\mathbf{P}$ operations. The set of all links in α_i forms a *chain* $\pi_i = (t^1_{r_1}, t^1_{r_2})(t^1_{r_2}, t^1_{r_3}) \ldots (t^1_{r_{j-1}}, t^1_{r_j})$, where $ncr_{m(i)} \leq r_1 < r_2 < \ldots < r_j < safe_{m(i)}$. Let $\mathfrak{B}_{m(i)}$ be the set of all distinct links in π_i.

Now, by the construction in the proof of Theorem 3.17, given $c \geq 0$ there exists $i \geq 0$ such that the length of the chain π_i is greater than or equal to c. If we define $\mathfrak{B} = lim\ sup\ \mathfrak{B}_{m(i)}$ as $i \dashrightarrow \infty$, then there exists $C \geq 0$ such that $i \geq C$ implies that π_i is composed solely of links in $\mathfrak{B}$. Because the number of $\mathbf{P}$ operations in α_i is a monotonically increasing function of i, this means that chains of arbitrary length may be formed from links in $\mathfrak{B}$. However, since the total number of transitions in $\mathfrak{T}'_{\Gamma}$, and hence the total number of distinct links is finite, it follows that $\mathcal{B}$ must contain a *cycle* $\chi = (u_0, u_1)(u_1, u_2) \dots (u_{k-1}, u_0)$, of length k. In the sequel, suppose all subscripts of u to be reduced modulo k; that is, let $u_i = u_{i \bmod k}$ for all i.

The remainder of the proof consists of the following:

(1) From among the numbers $m(0)$, $m(1)$, ... , select $n(0) < n(1) < \dots$, such that $u_0 \in \mathcal{A}_{n(0)}$ and for each $i > 0$ the link (u_i, u_{i+1}) is in $\mathcal{B}_{n(i+1)}$.

(2) Select a_1, a_2, ... , and b_0, b_1, ... , with $ncr_{n(0)} \leq b_0 < safe_{n(0)}$ and $ncr_{n(i)} \leq a_i < b_i < safe_{n(i)}$ for each $i > 0$, such that $t^1_{a_i} = u_i$, $t^1_{b_i} = u_{i+1}$, and there are no **P** operations in $seq(\alpha, a_i{+}1, b_i)$. That is, for each $i > 0$, $t^1_{a_i}$ and $t^1_{b_i}$ are the endpoints of the ith link (modulo k) in the cycle χ.

We will then show that the sequence

$$\beta = seq(\alpha, 0, ncr_{n(0)})alt(\alpha, ncr_{n(0)}, b_0)seq(\alpha, b_0, a_1)$$
$$alt(\alpha, a_1, b_1)seq(\alpha, b_1, a_2) \dots\ alt(\alpha, a_i, b_i)seq(\alpha, b_i, a_{i+1}) \dots$$

is an execution sequence for Γ that starves process 2. The situation is diagrammed in Figure 3.3.

The selection of the a_i, b_i, and $n(i)$ is performed inductively as follows:

Base: Choose (from among the $m(i)$) $n(0) \geq C$ such that $u_0 \in \mathcal{A}_{n(0)}$, and define b_0 so that $ncr_{n(0)} \leq b_0 < safe_{n(0)}$ and $t^1_{b_0} = u_0$. Since $b_0 < safe_{n(0)}$, we may use Lemma 3.15 to show that

$$\beta_0 = seq(\alpha, 0, ncr_{n(0)})alt(\alpha, ncr_{n(0)}, b_0)$$

is an initial execution sequence for Γ, and Corollary 3.20 to show that $seq(\alpha, b_0, \infty)$ is an execution sequence from $nxt(q_1, \beta_0)$. If $q'_{b_0} = nxt(q_1, \beta_0)$, then process 1 is at

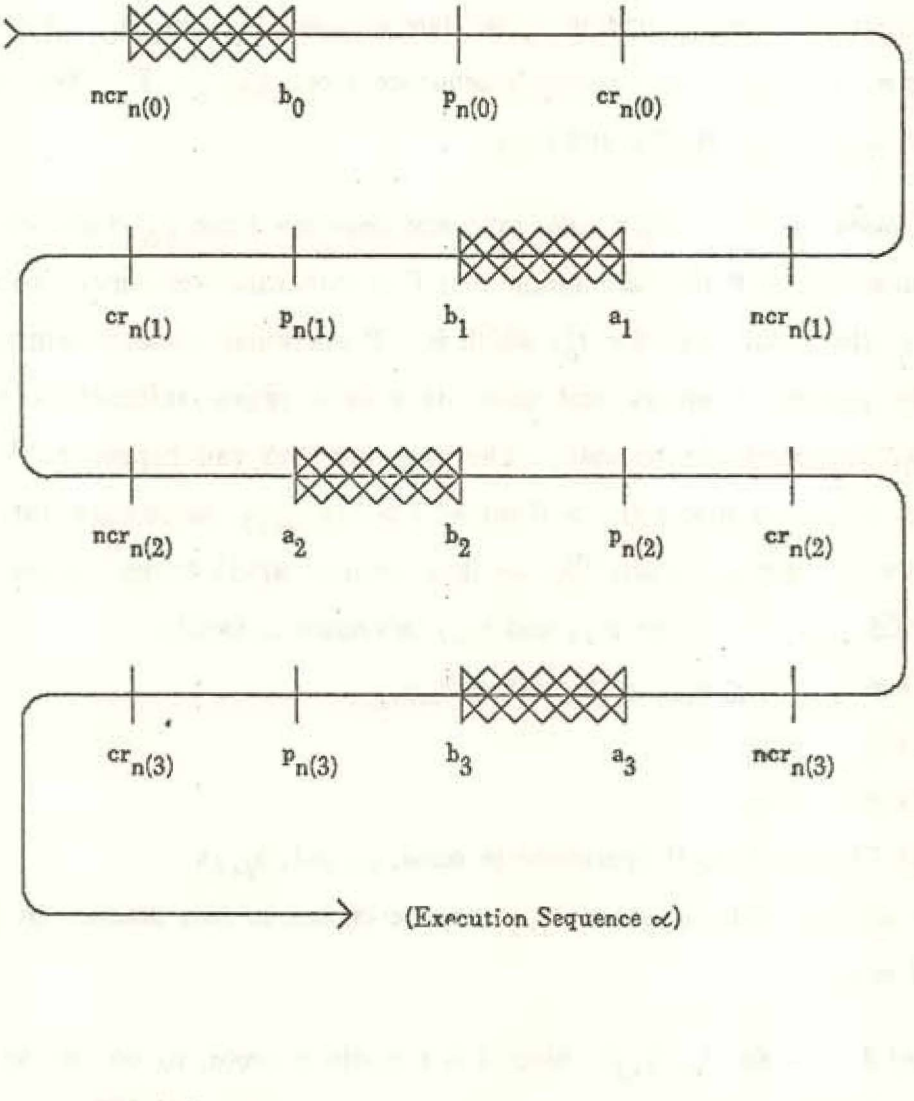


Figure 3.3 - Diagram for Proof of Theorem 3.21

the input place for transition $t^1_{b_0}$ in state q'_{b_0}, and $q'_{b_0}\ {}_1{=}_2\ q'_{b_0}$.

Induction Step: Suppose, for some $i \geq 0$, we have chosen $n(i)$ and b_i such that $ncr_{n(i)} \leq b_i < safe_{n(i)}$, and $t^1_{b_i} = u_i$. Suppose further that we have shown that β_i is an initial execution sequence for Γ, such that if $q'_{b_i} = nxt(q_I, \beta_i)$, then process 1 is at the input place for transition $t^1_{b_i}$ in state q'_{b_i}, and $q'_{b_i}\ {}_1{=}_2\ q'_{b_i}$. Finally, suppose that $seq(\alpha, b_i, \infty)$ is an execution sequence from q'_{b_i} for Γ. We now wish to construct a_{i+1}, b_{i+1}, $n(i+1)$, and β_{i+1}.

If $seq(\alpha, b_i, \infty)$ were a *valid* execution sequence from q'_{b_i}, then we would have a contradiction with the assumption that Γ is starvation-free, since process 2 would be left at the input place for $t^2_{b_i}$, which is a **P** operation on some semaphore, call it s_i, while process 1 enters and exits its critical region infinitely often. Hence $seq(\alpha, b_i, \infty)$ must not be valid. The only way this can happen is if there exists $n(i+1) > n(i)$ such that $q_j(s_i) > 0$ for all $j \geq ncr_{n(i+1)}$. Since each link in the cycle χ appears in infinitely many $\mathcal{B}_i$, we may assume $n(i+1)$ to be chosen so that $(u_i, u_{i+1}) \in \mathcal{B}_{n(i+1)}$. Now, let a_{i+1} and b_{i+1} be chosen to satisfy:

(1) $ncr_{n(i+1)} \leq a_{i+1} < b_{i+1} < safe_{n(i+1)}$

(2) $t^1_{a_{i+1}} = u_i$

(3) $t^1_{b_{i+1}} = u_{i+1}$

(4) There are no **P** operations in $seq(\alpha, a_{i+1}+1, b_{i+1})$

We are assured that a_{i+1} and b_{i+1} may be chosen in this manner by the way we selected $n(i+1)$.

Let $\delta = seq(\alpha, b_i, a_{i+1})$. Since δ is a prefix of $seq(\alpha, b_i, \infty)$ we know that δ is an execution sequence from q'_{b_i}. Let $q'_{a_{i+1}} = nxt(q'_{b_i}, \delta)$. Now, in state $q'_{a_{i+1}}$ process 1 is at the input place for transition $t^1_{a_{i+1}}$, which is a **P** operation on s_i. Also, process 2 is at the input place for transition $t^2_{b_i}$. But $t^1_{b_i} = t^1_{a_{i+1}} = u_i$ by construction, and since Γ has no local variables, this means that $q'_{a_{i+1}}\ {}_1{=}_2\ q'_{a_{i+1}}$.

By the way $n(i+1)$ was chosen, we know that $q'_{a_{i+1}}(s_i) - ind(s_i, t^1_{a_{i+1}}) > 0$. Hence $q'_{a_{i+1}}(s_i) \geq 2$. The sequence $t^1_{a_{i+1}} t^2_{a_{i+1}}$ is therefore an execution sequence from $q'_{a_{i+1}}$. Since the sequence $alt(\alpha, a_{i+1}+1, b_{i+1})$ contains no **P** operations, Lemma 3.15 shows that $alt(\alpha, a_{i+1}+1, b_{i+1})$ is an execution sequence from $q'_{a_{i+1}}$. Define $\beta_{i+1} = \beta_i \delta alt(\alpha, a_{i+1}, b_{i+1})$.

It is not difficult to see that if $q'_{b_{i+1}} = nxt(q_1, \beta_{i+1})$, then for all $s \neq s_i$ and all $j \geq b_{i+1}$, $q'_{b_{i+1}}(s) - ind(s, b_{i+1}, j) = q'_{b_i}(s) - ind(s, b_i, j) \geq 0$. Also, by the way $n(i+1)$ was chosen, $q'_{b_{i+1}}(s_i) - ind(s_i, b_{i+1}, j) = q'_{b_i}(s_i) - ind(s_i, b_i, j) - 1 \geq 0$ for all $j \geq b_{i+1}$. Hence Lemma 3.16 shows that $seq(\alpha, b_{i+1}, \infty)$ is an execution sequence from $q'_{b_{i+1}}$. This completes the induction step.

Thus we have proved that β is an execution sequence for Γ. In addition, β is *valid* since processes 1 and 2 each have infinitely many transitions in β. However, process 2 remains forever in the trying region while process 1 enters and exits its critical region infinitely often. Hence β starves process 2, a contradiction, and we conclude that Γ must have local variables. ▩

3.9 Conclusion

In this chapter we presented a number of results that shed some light on the "power" of weak semaphore primitives, when used to implement starvation-free mutual exclusion. The "negative" results of this chapter are most interesting when viewed in relation to the complementary "positive" results to be presented in the next chapter. Let us proceed now to the statement of these results.

4. Some Existence Results

In the last chapter we examined various classes of mutual exclusion systems, and were able to show that within some of these classes, weak semaphores are not sufficient to implement starvation-free mutual exclusion. In particular, there are no solutions to the starvation-free mutual exclusion problem that use weak general semaphores, in the class *NBW* ∩ *SYM* ∩ *NM*, and there are no solutions using weak binary semaphores even in the larger class *SYM* ∩ *NM*. To obtain a more complete picture of the situation, we should ask whether there are any solutions at all using binary or general, weak or blocked-set semaphores, in any of the restricted classes of mutual exclusion systems we have been examining. The answers are interesting, and are the subject of this chapter.

This chapter is organized as a series of "examples", in which a candidate solution to the starvation-free mutual exclusion problem is presented, along with an informal argument that it has the stated properties. Only one of these solutions (Example 4.1) will actually be proved correct, and the reason is that the correctness proof for just this one example requires all of Chapter 5 and the Appendix.

> **Example 4.1** - The mutual exclusion system described by Morris' program in Figure 2.3 is a solution to the starvation-free mutual exclusion problem which is in *NBW* ∩ *SYM* ∩ *NM*. Although the solution is shown using

> blocked-set general semaphores, blocked-set binary semaphores work just as well.

Discussion - Note that this solution, combined with Theorems 3.14 and 3.17, shows that either blocked-set binary or blocked-set general semaphores are more "powerful" than weak semaphores, as far as their ability to implement starvation-free mutual exclusion is concerned.

A proof of correctness for this solution is the subject of Chapter 5 and the Appendix. Note that such a proof requires that the algorithmic presentation in Figurė 2.3 be translated into graphical form, and this is done in Chapter 5. It should be possible, however, for the reader to mentally verify that the algorithmic presentation of Figure 2.3 indeed represents a mutual exclusion system in *NBW* ∩ *SYM* ∩ *NM*. No graphical translations will be provided for any of the other examples of this chapter.

> **Example 4.2** - The mutual exclusion system described by Figure 4.1 is a solution to the starvation-free mutual exclusion problem which is in *NBW*. Although the solution is shown using weak binary semaphores, in fact it makes no difference what type of semaphores are used.

Discussion - Note that this solution is *not* in *SYM*, since explicit use is made of the process number *i*. Example 4.3, though, shows how this solution may be transformed through the addition of one semaphore variable and one local variable in each process, to a solution that *is* in *SYM*, but *not* in *NM*.

The solution works as follows: When process *i* enters the trying region, the first thing it does is to indicate its presence by setting the variable *flags*[*i*] to **true**. The process next tests the global variable *empty* to find out if any other processes are in the trying or critical regions. If *empty* is **true**, indicating that there are no other processes, process *i* remembers this by setting its local variable *first* to **true**. It then proceeds directly to the critical region. If there are other processes, *first* remains **false**, and process *i* executes a **P**(*sem*[*i*]) to await its turn in the critical region.

When process *i* enters the leaving region, it resets *flags*[*i*] to **false**, indicating that it is no longer interested in executing in the critical region. It then examines the array *flags* to determine the next process to execute in the critical region. The global variable *next* is used as a roving pointer to ensure the fairness of this selection. The selected process is signalled via a **V**(*sem*[*j*]) operation, and process *i* exits the leaving region. If all the entries of *flags* are **false**, no process is presently interested in executing a critical region, and the variable *empty* is reset to **true**.

The semaphore *mutex* has a dual purpose. First, it ensures mutually exclusive access to the variable *empty*. Second, it prevents the possibility of the deadlock situation where a process in the leaving region finds all entries of *flags* to be **false**, but before it can set *empty* to **true**, a process in the trying region reads it, and then waits at the **P**(*sem*[*i*]) for a **V**(*sem*[*i*]) that may never occur.

It should not be difficult to convince oneself that the correctness of the solution does not depend upon what type of semaphores are used.

> **Example 4.3** - Any solution to the starvation-free mutual exclusion problem may be transformed into a symmetric solution through the addition of a single semaphore variable and a local variable in each process, as shown in Figure 4.2. The resulting solution, however, is *not* in *NM*.

Discussion - In particular, applying this transformation to the solution of Figure 4.1 shows that there is a solution in $NBW \cap SYM$, using any type of semaphores.

```
        global integer next initially 1;
        global boolean empty initially true;
        global boolean array flags[1:N] initially false;
        weak binary semaphore mutex initially 1;
        weak binary semaphore array sem[1:N] initially 0;

(Program Executed by Process i)

        local boolean first;
        local integer j;
loop:   <noncritical region>
        flags[i] := true;
        first := false;
        P(mutex);
        if empty then begin
                empty := false;
                first := true
        end;
        V(mutex);
        if ¬first then P(sem[i]);
        <critical region>
        flags[i] := false;
        P(mutex);
        for j = next + 1 to N step 1, 1 to next step 1 do
                if flags[j] then begin
                        next := j;
                        V(sem[j]);
                        goto out
                end;
        empty := true;
out:    V(mutex);
        goto loop;
```

Figure 4.1 - Asymmetric Solution With No Busy-Waiting

Figure 4.2 shows how the *N* trying regions and *N* leaving regions of the given solution may be combined into a single trying and leaving region that is run by all processes in the transformed solution. The semaphore *mutex1* is shown as a weak binary semaphore in Figure 4.2 to be consistent with Figure 4.1, however when applying this transformation to an arbitrary solution, the semaphore *mutex1* should be of the same type as the semaphores in the given solution.

The transformed solution operates as follows: The first time each process enters the trying region, its local variable *ident* has the value zero. The process then "picks a number" by assigning the value of *procno* to *ident*, and then incrementing *procno*. Each process thus obtains a unique value of *ident*, which it then uses to select among the *N* different trying and leaving region protocols. It is important to note that "picking a number" occurs exactly once per process, the first time that process enters the trying region, and is not performed on subsequent visits to the trying region. Thus even though the semaphore *mutex1* is a weak semaphore, there is no possibility of starvation.

It is easy to see that the transformation does not introduce the possibility of busy-waiting. Since the program text run by one process is identical to that run by any other process, the transformed solution is symmetric. The only thing that distinguishes one process from another is the value of the variable *ident*. The correctness of this solution depends crucially on the fact that it uses "memory". As in Example 4.2, it is not difficult to see that the correctness of the transformed solution does not depend upon the type of semaphores used.

Example 4.4 - The mutual exclusion system described by Figure 4.3 is a solution to the starvation-free mutual exclusion problem for two processes, which uses weak general semaphores, is in *SYM* ∩ *NM*, but is *not* in *NBW*.

```
        global integer procno initially 1;
        weak binary semaphore mutexI initially 1;

(Program Executed by Process i)

        local integer ident initially 0;

loop:   <noncritical region>
        if ident = 0 then begin
                P(mutexI);
                ident := procno;
                procno := procno + 1;
                V(mutexI)
        end;
        if ident = 1 then <Execute Trying Region for Process 1>
        else if ident = 2 then <Execute Trying Region for Process 2>
                .
                .
                .
        else if ident = N then <Execute Trying Region for Process N>
        <critical region>
        if ident = 1 then <Execute Leaving Region for Process 1>
        else if ident = 2 then <Execute Leaving Region for Process 2>
                .
                .
                .
        else if ident = N then <Execute Leaving Region for Process N>
        goto loop;
```

Figure 4.2 - Construction of a Symmetric Solution From an Asymmetric Solution

Discussion - Note that this result, coupled with Theorem 3.14, shows a sense in which weak general semaphores are strictly more "powerful" than weak binary semaphores. Dijkstra [DIJKS66] claims that general semaphores are "superfluous" given binary semaphores as primitives. At least for weak semaphores, if we are concerned about the properties of symmetry and no memory in our solution, this is not the case. This also explains why any attempt to implement weak general semaphores with weak binary semaphores, such as that in [SHAW74], must either introduce asymmetry, violate the "no memory" property, or fail.

The operation of the solution in Figure 4.3 may be visualized in the following way: Think of the semaphore variable *sem* as being a bag that contains a number of pebbles. If both processes are in the noncritical region, then the number of pebbles in the bag is $2 \cdot cycle - 1$. To enter the critical region, a process must remove *cycle* pebbles from the bag by executing *cycle* **P**(*sem*) operations in the trying region. Mutual exclusion is therefore ensured, since there is always one less than the number of pebbles needed for both processes to enter the critical region simultaneously. In the leaving region, a process increments *cycle* and then returns $cycle + 1$ pebbles to the bag. Thus, each time the leaving region is executed, *cycle* is increased by one, and the total number of pebbles is increased by two. Note that the fact that the total number of pebbles is constantly increasing ensures that a process may not starve at the weak **P**(*sem*) operation in the trying region while the other process repeatedly executes critical regions.

Ignoring for the moment the **P**(*wait*) operation in the trying region, since a process, for example process 1, cannot starve at the **P**(*sem*) operation while process 2 repeatedly executes critical regions, the only way for process 1 to remain forever in the trying region is if it continually loops back to *tloop*. Since process 1 removes a pebble from the bag each time around the loop, process 2 must be repeatedly adding pebbles to the bag and also repeatedly incrementing *cycle.* Now, when process 1 entered the trying region, it recorded the value of *cycle* in its local variable *savecycle.* Each time around the loop, it checks to see if the current value of *cycle* exceeds *savecycle* by more than *LIMIT.* If so, then process 2 has gotten "too far ahead" of process 1. Upon noticing this, process 1 sets the global variable *haltflag* and its local variable *lhalt* to one. When process 2 sees that *haltflag* is one, instead of continuing to execute critical regions, it performs a **P**(*wait*). Process 1 is now free to enter the critical region at will. The local variable *lhalt* is used so that process 1 does not itself become halted because *haltflag* is one. When process 1 executes the leaving region, it resets *haltflag* to zero, and signals process 2 with a **V**(*wait*).

The interesting and novel feature of this solution is the way in which continually increasing the number of pebbles in the bag prevents a process from starving at a weak **P** operation. Note that this means that no *a priori* bound may be placed on the length of time a process may spend in the trying region; indeed, as the system ages, the length of the trying and leaving region protocols becomes longer and longer! Thus this solution is not in *NBW,* although it is in *SYM* ∩ *NM.*

```
        weak general semaphore sem,wait initially 1, 0;
        global cycle, haltflag initially 1, 0;
```

(Program Executed by Process i $(1 \leq i \leq 2)$)

```
        local savecycle, count, lhalt initially 0, 0, 0;

loop:   <noncritical region>
        savecycle := cycle;
tloop:  P(sem);
        count := count + 1;
        if count = cycle then goto crit;
        if lhalt = 1 then goto tloop;
        if cycle − savecycle > LIMIT then begin
          haltflag := 1;
          lhalt := 1
        end;
        if haltflag = 1 ∧ lhalt = 0 then P(wait);
        goto tloop;
crit:   <critical region>
        cycle := cycle + 1;
        if lhalt = 1 then begin
          lhalt := 0;
          haltflag := 0;
          V(wait)
        end;
        for savecycle := cycle + 1 step -1 until 1 do V(sem);
        count := savecycle := 0;
        goto loop;
```

Figure 4.3 - Two-Process Solution with Weak General Semaphores

5. A Correctness Proof

In Example 4.1, the mutual exclusion system of Figure 2.3 was proposed as a blocked-set general semaphore solution, in $NBW \cap SYM \cap NM$, to the starvation-free mutual exclusion problem. In this chapter this solution will be proved correct. It is left to the reader to verify that if the blocked-set general semaphore operations are replaced with blocked-set binary semaphore operations, then the arguments to be presented in this chapter hold essentially unchanged.

The most outstanding feature of the proof is its length, and this is the reason why correctness proofs for other solutions proposed in this thesis have been omitted. It is surprising what lengthy formulas seem to be necessary to prove a handful of properties about a short program. The presentation of the proof has been guided by the philosophy that, as much as possible, the portions of the proof that actually contain intuition about the operation of the solution should not be obscured by tedious details of verification that might well be performed by a machine. For this reason, the proof has been divided between this chapter and the Appendix. This chapter contains the portions of the proof in which intuition is required, and the remaining mechanical details are given in the Appendix.

Due to a lack of well-established techniques for proving properties of parallel programs, a detailed discussion of the three techniques we will be using is prerequisite to the proof itself. The three techniques are the *induction principle*, the *well-founded set* method, and the *parallel program homomorphism* method. The induction principle and the well-founded set method are adaptations of techniques used in [KELLE76] and [KWONG78]. The notion of a parallel program homomorphism is hinted at in [DOEPP76]. Parallel program homomorphisms allow us to abstract away from irrelevant details, and it is this property that permits a clean separation of the mechanical verification parts of the proof from the more important, intuitive parts.

The outline for this chapter is as follows: Before we may prove anything about the mutual exclusion system of Figure 5.1, we must first perform a translation from the algorithmic notation of Figure 5.1 to the graphical notation of the mutual exclusion system model. The induction principle will be used to prove that the system has the mutual exclusion property, and the well-founded set method will be used to prove the system free from indefinite postponement. The proofs that the system has the mutual exclusion property and is free from indefinite postponement are quite easy. To verify that the system is deadlock-free and starvation-free is much harder. Freedom from deadlock is proved with the help of a rather lengthy *invariant*, which is a predicate true for all reachable states. Intuition must be used to construct this invariant, however once constructed, the actual proof of invariance is strictly mechanical, and is left to the Appendix. To simplify the proof that the system is starvation-free we will use a parallel program homomorphism to eliminate details that obscure the properties of interest.

5.1 Informal Discussion of Morris' Solution

Before proceeding with the formal discussion, let us examine informally the operation of the system of Figure 5.1. Execution of this system proceeds in two phases. In phase one, the semaphore *m* is zero, and processes exit the noncritical region, enter the trying region, and filter down to the **P**(*m*) operation, where they become blocked. In phase two, the semaphore *a* is zero; preventing new arrivals

```
blocked-set general semaphore a, b, m initially 1, 1, 0;
global count1, count2, initially 0, 0;

(Program Executed by Process i)

        loop:   <noncritical region>
                P(b);
                count1 := count1 + 1;
                V(b);
                P(a);
                P(b);
                count1, count2 := count1 − 1, count2 + 1
                if count1 > 0 then begin
                        V(b);
                        V(a)
                end else begin
                        V(b);
                        V(m)
                end;
                P(m);
                <critical region>
                count2 := count2 − 1;
                if count2 > 0 then V(m) else V(a);
                goto loop;
```

Figure 5.1 - Morris' Solution to the Starvation-Free Mutual Exclusion Problem

from passing the **P**(a) operation in the trying region. Processes waiting at the **P**(m) operation are allowed to enter the critical region one-by-one. When there are no more processes waiting at the **P**(m), phase one begins again. Thus mutual exclusion is maintained by the semaphore m.

The most interesting aspect of this solution is the way in which freedom from starvation is achieved. The key ideas are the following: (1) As long as processes continue to execute, the system will continue to alternate between phases one and two; (2) The system can only switch from phase one to phase two if the set of processes blocked on the semaphore b has recently been emptied. System execution changes from phase one to phase two at the instant a process at the conditional statement in the trying region discovers that $count1 = 0$. The hardest part of the solution to understand is why a process discovering that $count1 = 0$ may conclude that the set of processes blocked on the semaphore b has recently been emptied, and hence that it is safe to allow processes to execute in the critical region.

5.2 Morris' Solution Presented in the Parallel Program Model

Figure 5.9 at the end of this chapter displays the graph for the ith process in the mutual exclusion system that results when the program of Figure 5.1 is translated into graphical form. Throughout this chapter, we will refer to this mutual exclusion system as Σ.

The set $\mathcal{G}_\Sigma$ of global variables for Σ contains two variables, *count1* and *count2*. There are nine synchronization variables in $\mathcal{S}_\Sigma$: the three semaphore variables a, b, m, and the corresponding set variables *a_bl*, *a_en*, *b_bl*, *b_en*, *m_bl*, *m_en*. The ith process has one local variable, *loc_i*. The initial state for Σ assigns the value zero to all local variables, all global variables, and semaphore variable m. Semaphore variables a and b have initial value one, and all set variables have initial value $\varnothing$.

The graph for the *i*th process in Σ contains $34 + 7N$ transitions, where N is the number of processes in the system. However, for our purposes it will never be necessary to distinguish between the N transitions that model the nondeterministic selection of a process from the blocked set in a V operation. For this reason, in each V operation, these N transitions have all been given the same name. The names for the transitions in the graph for process *i* in Σ are: NCT_*i*, TT1_*i*, ... , TT30_*i*, CT_*i*, and LT1_*i*, ... , LT9_*i*. Transitions NCT_*i* and CT_*i* connect the noncritical region to the trying region, and the critical region to the leaving region, respectively. Transitions TT1_*i*, ... , TT30_*i* are the trying region transitions ("TT" stands for "trying transition") and transitions LT1_*i*, ... , LT9_*i* are the leaving region transitions ("LT" for "leaving transition"). Because of the symmetry of the system, we will usually not be interested in distinguishing between transitions in different processes. It will therefore be convenient to speak, for example of "transition TT3", as if there were only one such transition, rather than one in each process. Used in this way, the phrase "transition TT3" means, roughly, "transition TT3_*i*, for some process *i*".

Due to space considerations, statement labels have been omitted from transitions LT4_*i*, LT5_*i*, LT7_*i*, and LT8_*i* in Figure 5.9(h). However, since these transitions are part of V operations, it should be obvious what the labels should be.

There are twenty-seven places in the *i*th process. The noncritical place is NCP_*i* and the critical place is CP_*i*. The trying region places are numbered TP1_*i*, ... , TP20_*i*, and the leaving region places are LP1_*i*, ... , LP5_*i*. It will be convenient to assign names to various important sets of places, and this is done in Figure 5.2. The names of these sets are supposed to be somewhat mnemonic; their meanings are as follows: *NCR* and *CR* stand for "noncritical region" and "critical region", respectively. *TPi* is the *i*th place in the trying region, and *LPi* is the *i*th place in the leaving region. *TR* and *LR* stand for "trying region" and "leaving region", respectively. *MUTEX1* and *MUTEX2* stand for "mutually exclusive regions" one and two. It will be seen later that at most one process at a time can be at a place in the set *MUTEX1*, and similarly for *MUTEX2*. *WAITA* and *WAITM* are the sets of places where processes wait to pass the $\mathbf{P}(a)$ and $\mathbf{P}(m)$ operations, respectively, and *WAITB1* and *WAITB2* are the sets of places where processes wait to pass the two $\mathbf{P}(b)$ operations.

It will be convenient to introduce the following notation: If E is an expression involving variables or names of places in a parallel program, then let $[E](q)$ denote the value obtained by evaluating the expression E in state q. For example, if $q(a) = 1$ and $q(b) = 2$, then $[a + b](q) = 3$. If $\mathbb{S}$ is a set of places, and i is a process number, then define the predicate $[i \text{ at } \mathbb{S}](q)$ to be true iff the token for process i is at a place in $\mathbb{S}$ in state q. Also, let $[\,|\mathbb{S}|\,](q)$ denote the total number of tokens residing at places in $\mathbb{S}$ in state q. Let $en(a)$ abbreviate the expression "$a > 0 \vee a\_en \neq \varnothing$", and similarly for $en(b)$ and $en(m)$. Note that if $[en(a)](q)$ is true, then the semaphore a is "enabled" in state q in the sense that either a process at the blocked-place for a $\mathbf{P}(a)$ is enabled to proceed, or a process arriving at the input place to a $\mathbf{P}(a)$ may complete the $\mathbf{P}$ operation without becoming blocked. Finally, if ν is a set variable, then let $[card(\nu)](q)$ be the number of elements in the set $q(\nu)$.

5.3 Mutual Exclusion

Now that the problem of nomenclature is out of the way, we may begin proving properties of Σ. The first proof technique we will use is called the *induction principle*, for proving *invariant assertions* about the state of a parallel program. An assertion is simply a predicate on states. If Γ is a parallel program, and if P is a predicate over $\mathbb{Q}_{\Gamma}$, then we say that P is *invariant* for Γ iff $P(q)$ is true for any reachable state q. The induction principle is presented here in a form essentially identical to that of [KELLE76]; where it is argued that the induction principle for

$$NCR = \{\mathrm{NCP\_1, NCP\_2, \ldots , NCP\_}N\}$$

$$CR = \{\mathrm{CP\_1, CP\_2, \ldots , CP\_}N\}$$

$$TPi = \{\mathrm{TP}i\mathrm{\_1, TP}i\mathrm{\_2, \ldots , TP}i\mathrm{\_}N\}, \text{ for } 1 \leq i \leq 20$$

$$LPi = \{\mathrm{LP}i\mathrm{\_1, LP}i\mathrm{\_2, \ldots , LP}i\mathrm{\_}N\}, \text{ for } 1 \leq i \leq 5$$

$$TR = \cup_{i=1}^{20} TPi$$

$$LR = \cup_{i=1}^{5} LPi$$

$$MUTEX1 = \cup_{i=3}^{5} TPi$$

$$MUTEX2 = \cup_{i=8}^{15} TPi \cup TP17$$

$$WAITA = TP6 \cup TP7$$

$$WAITM = TP19 \cup TP20$$

$$WAITB1 = TP2$$

$$WAITB2 = TP8 \cup TP9$$

Figure 5.2 - Useful Names for Various Subsets of $\mathcal{P}_{\Sigma}$

proving properties of parallel programs is a natural generalization of the *inductive assertion* technique of [FLOYD67] for proving properties of sequential programs.

> **Induction Principle** - Let Γ be a parallel program with initial state q_I, and let P be a predicate over $\mathfrak{Q}_\Gamma$. If $P(q_I)$ is true, and if $P(q)$ implies $P(nxt(q, t))$ for all $q \in \mathfrak{Q}_\Gamma$ and all $t \in \mathfrak{I}_\Gamma$ enabled in state q, then P is satisfied by any reachable state of Γ.

We can apply the induction principle immediately to prove that Σ has the mutual exclusion property. In this and subsequent discussions, phrases such as "$[B](q)$ is true" and "B is true in state q" will be used interchangeably, to avoid unpleasant repetition.

> **Claim 5.1** - The system Σ has the mutual exclusion property.

Proof - Let *sum* abbreviate the expression $|CR| + |LR|$. Let $excl \equiv excl_1 \wedge excl_2 \wedge excl_3$, where

$$excl_1 \equiv \ (sum > 0 \rightarrow \neg en(m))$$
$$excl_2 \equiv \ sum \leq 1$$
$$excl_3 \equiv \ m \leq 1 \wedge card(m\_en) \leq 1 \wedge (m > 0 \rightarrow m\_en = \varnothing)$$

Then we claim that $[excl]$ is invariant for Σ. Note that if this is true, then Σ has the mutual exclusion property, since the invariance of $[excl]$ implies the invariance of $[CR \leq 1]$. The invariance of $[excl]$ will be proved by the induction principle. Note that we are unable to use the induction principle to prove the invariance of $[|CR| \leq 1]$ directly; we have had to strengthen the predicate before the proof will go through.

It is obvious that $[excl](q_I)$ is true. Let q be a state satisfying $[excl]$, let t be a transition enabled in state q, and define $q' = nxt(q, t)$. We must show that $[excl](q')$ is true. Consider first $excl_1$. Since $[excl](q)$ is true, then the only transitions t that could possibly make $excl_1$ false in state q' are TT28 and TT30. However, since $excl_3$ is true in state q, firing TT28 or TT30 from state q is guaranteed to make $en(m)$ false in the resulting state q'. Hence $excl_1$ is true in state q'.

Now, $excl_2$ can only be false in state q' if $sum = 1$ in state q and TT28 or TT30 is fired. But since $excl_1$ is true in state q, $[sum = 1](q)$ implies $[\neg en(m)](q)$, which means that TT28 and TT30 cannot be enabled when $sum = 1$.

Finally, $excl_3$ can be false in state q' only if transition LT4 or LT5 is fired from state q. However, for either of these transitions to be enabled in state q, we must have $sum = 1$ in state q and hence $[\neg en(m)](q)$ must be true. Now, $[\neg en(m)](q)$ is by definition true iff $[m = 0 \wedge m\_en = \varnothing](q)$ is true. Therefore, firing LT4 or LT5 causes either $[m = 1](q')$ to be true, or $[card(m\_en) = 1](q')$ to be true, respectively. In either case $excl_3$ is true in state q' and the Claim is proved. ▩

The predicate $[excl]$ was rather simple, and the proof of its invariance was easy, since it was easy for us to see that only a few transitions in Σ could affect its truth. To prove that Σ is deadlock- and starvation-free will require a much more complicated predicate, the invariance of which implies the invariance of $excl$. In the Appendix we will see that nearly all transitions in Σ can affect the truth of this predicate, and it therefore would be better if the large number of cases were checked by machine, instead of by hand.

5.4 Freedom From Indefinite Postponement

It was noted that the induction principle is a natural parallel program generalization of the inductive assertion technique of [FLOYD67] for proving properties of sequential programs. The well-founded set method presented in this section can be considered a natural parallel program generalization of Floyd's technique for proving termination of sequential programs. The version presented here is not only useful for proving that an execution sequence is finite, but may be applied to infinite execution sequences as well. For example, we will show that no process in a system of processes may fire an infinite number of transitions without leaving an acyclic part of its graph. This in turn will be used to show that Σ is free from indefinite postponement.

Versions of the well-founded set method similar to this one may be found in [KELLE76] and [KWONG78].

> **Well-Founded Set Method** - Let Γ be a parallel program and $(\mathcal{W}, <)$ a well-founded set (one that has no infinite decreasing chains). Let α be a finite or infinite execution sequence for Γ, with corresponding state sequence $q_0q_1 \ldots$. Let $f\colon \mathfrak{Q}_\Gamma \dashrightarrow \mathcal{W}$, and P be a predicate over $\mathfrak{Q}_\Gamma$ such that for all states q, and all transitions t enabled in state q, if $P(q)$ and $P(nxt(q, t))$ are true then $f(nxt(q, t)) \leq f(q)$. Let $\mathfrak{U}$ be the set of all $j \geq k$ such that $f(q_{j+1}) < f(q_j)$. Then either $\mathfrak{U}$ is finite, or there exists $j \geq k$ such that $P(q_j)$ is false.

An important special case that we shall encounter is when $(\mathcal{W}, <)$ is the natural numbers under the usual ordering.

As an application of the well-founded set method, we will now prove that Σ is free from indefinite postponement. We will make use of the following obvious result, for which a proof is given only to illustrate the use of the well-founded set method.

> **Observation 5.2** - Let Γ be a system of N processes, and let Ξ be an acyclic subgraph of the graph for the ith process. Then there is no infinite execution sequence α from some state q_0 for Γ, containing infinitely many transitions for process i, such that if $q_0 q_1 \ldots$ is the state sequence corresponding to α, then for all $j \geq 0$ the token for process i is at a place in Ξ in state q_j.

Proof - Suppose there were such a sequence α. Let $\mathcal{W} = \mathcal{P}^{i}_{\Gamma}$, the set of places in the graph for process i. Since Ξ is a directed acyclic graph, we may define an ordering $<$ on $\mathcal{W}$ as follows: If p and p' are places in Γ, then

(1) $p < p'$ if p' is a predecessor of p in Ξ.

(2) p and p' are incomparable under $<$ if either p or p' is not in Ξ.

Let $P(q)$ be true iff the token for process i is at a place in Ξ in state q, and define $f(q) = p$ if the token for process i is at place p in state q. It is clear, if q is a state, t is a transition for process i enabled in state q, and $q' = nxt(q, t))$, that $P(q)$ and $P(q')$ imply that $f(q') < f(q)$. By the well-founded set method, if α contains infinitely many transitions for process i, then there exists $j \geq k$ such that $P(q_j)$ is false. This is a contradiction and the Observation is proved. ▇

> **Claim 5.3** - Σ is free from indefinite postponement.

Proof - Suppose not. Then there is a valid infinite initial execution sequence α for

Σ, with corresponding state sequence $q_0 q_1 \ldots$, such that for some k and all $j \geq k$ no process is in the critical region in state q_j. Since the number of processes is finite, there must be some process i, such that α contains infinitely many transitions for process i. Now, by the structure of the graph for process i in a mutual exclusion system, a process that executes infinitely many transitions without entering the critical region must either remain forever in the trying region, or forever in the leaving region. But, since the trying and leaving regions of Σ are acyclic, process i may not remain forever in the trying region or forever in the leaving region, by the previous Observation. This is a contradiction, and the Claim is proved. ■

5.5 Freedom From Deadlock

We have seen that Σ has the mutual exclusion property and is free from indefinite postponement. In this section, we will show that Σ is deadlock-free. This will be done by showing that from any reachable state, we can construct an execution sequence that removes all processes from the trying and leaving regions. The key property we need for this is that in any reachable state of Σ with at least one process in either the trying or leaving region, there is some process in either the trying or leaving region that has an enabled transition.

To prove this property, we will prove the invariance of a rather lengthy predicate [*inv*], which encompasses most of the interesting statements about Σ that can be formulated as invariant assertions. The reason this predicate is so complicated is that it is not possible to use the induction principle to prove very many short statements of interesting properties of Σ. Instead, these short statements must be strengthened by conjoining additional terms before the proof will go through.

The construction of the invariant $[inv]$ is based on the following observation: the reachable states of Σ may be classified according to the configuration of tokens on the graph. Although the fact that there are twenty-seven places in the graph for the ith process implies that there are 27^N different configurations of tokens, for our purposes large classes of configurations may be regarded as "equivalent", reducing the total number of "different" configurations to thirteen. The classification of the reachable states into these thirteen categories was reached by a process of trial and error, requiring the use of intuition about the operation of the system Σ. A classification of states into less than thirteen categories is too weak for our purposes. The thirteen categories are defined by the predicates $[conf_i]$, for $1 \leq i \leq 13$, where the expressions $conf_i$ are given in Figure 5.3.

Note that the $[conf_i]$ are pairwise disjoint, meaning that no state can satisfy more than one of the $[conf_i]$. If we can show (which we will) that every reachable state q satisfies exactly one of the $[conf_i]$, then it is interesting to see that the mutual exclusion property of Σ is immediate. This is because a process can be in the critical region in state q only if $[conf_{12}](q)$ is true. However, this means that $|CR| = 1$ in state q, and hence there can never be more than one process in the critical region in any reachable state.

Corresponding to the predicates $[conf_i]$, which define the thirteen categories of reachable states, are thirteen additional predicates $[vbls_i]$, which describe the values of the variables of Σ in each of the categories. That is, if q is a reachable state that satisfies $[conf_i]$ for some i, then q also satisfies $[vbls_i]$. Stated another way, we assert the invariance of the predicate $[\vee_{i=1}^{13} (conf_i \wedge vbls_i)]$. The expressions $vbls_i$ are defined in Figure 5.4. The expression *procnotcounted* abbreviates $\exists i(i \text{ at } WAITB1 \wedge i \in b\_en)$. The quantifier used here, as well as any others we will use, ranges over process numbers, and is hence bounded by N.

$$\begin{aligned}
conf_1 \equiv\ & |MUTEX1| = 0 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 0 \wedge \\
& |CR| = 0 \wedge |LR| = 0 \\
conf_2 \equiv\ & |MUTEX1| = 1 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 0 \wedge \\
& |CR| = 0 \wedge |LR| = 0 \\
conf_3 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| > 0 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 1 \wedge \\
& |TP16| = 0 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \wedge |TP17| = 0 \\
conf_4 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| > 0 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge \\
& |TP16| = 1 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \\
conf_5 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| = 1 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge \\
& |TP16| = 1 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \\
conf_6 \equiv\ & |MUTEX1| = 1 \wedge |WAITA| > 0 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge \\
& |TP16| = 1 \wedge |TP18| = 0 \wedge |CR| = 0\ |LR| = 0 \\
conf_7 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| = 1 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge \\
& |TP16| = 0 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \\
conf_8 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| = 0 \wedge |WAITB2| = 1 \wedge |MUTEX2| = 0 \wedge \\
& |TP16| = 0 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \\
conf_9 \equiv\ & |MUTEX1| = 0 \wedge |WAITA| = 0 \wedge |WAITB2| = 0 \wedge |MUTEX2| = 1 \wedge \\
& |TP16| = 0 \wedge |TP18| = 0 \wedge |CR| = 0 \wedge |LR| = 0 \wedge |TP15| = 0 \\
conf_{10} \equiv\ & |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 1 \wedge \\
& |CR| = 0 \wedge |LR| = 0 \\
conf_{11} \equiv\ & |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 0 \wedge \\
& |WAITM| > 0 \wedge |CR| = 0 \wedge |LR| = 0 \\
conf_{12} \equiv\ & |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 0 \wedge \\
& |CR| = 1 \wedge |LR| = 0 \\
conf_{13} \equiv\ & |WAITB2| = 0 \wedge |MUTEX2| = 0 \wedge |TP16| = 0 \wedge |TP18| = 0 \wedge \\
& |CR| = 0 \wedge |LR| = 1
\end{aligned}$$

Figure 5.3 - Reachable States of Σ Classified Into Thirteen Categories

$$
\begin{aligned}
vbls_1 \equiv\ & (|WAITB2| = 0 \rightarrow (en(a) \wedge (|WAITA| = 1 \rightarrow procnotcounted))) \wedge \\
& (|WAITB2| = 1 \rightarrow (\neg en(a) \wedge (|WAITA| = 0 \rightarrow procnotcounted))) \wedge \\
& (|WAITM| > 0 \rightarrow (|WAITB1| + |WAITA| + |WAITB2| > 0)) \wedge \\
& en(b) \wedge \neg en(m)
\end{aligned}
$$

$$
\begin{aligned}
vbls_2 \equiv\ & (|WAITB2| = 0 \rightarrow en(a)) \wedge (|WAITB2| = 1 \rightarrow \neg en(a)) \wedge \neg en(b) \wedge \\
& \neg en(m)
\end{aligned}
$$

$$vbls_3 \equiv \neg en(a) \wedge \neg en(b) \wedge \neg en(m)$$

$$vbls_4 \equiv (|WAITA| = 1 \rightarrow procnotcounted) \wedge \neg en(a) \wedge en(b) \wedge \neg en(m)$$

$$vbls_5 \equiv \neg procnotcounted \wedge \neg en(a) \wedge en(b) \wedge \neg en(m)$$

$$vbls_6 \equiv \neg en(a) \wedge \neg en(b) \wedge \neg en(m)$$

$$vbls_7 \equiv \neg procnotcounted \wedge en(a) \wedge en(b) \wedge \neg en(m)$$

$$vbls_8 \equiv \neg procnotcounted \wedge \neg en(a) \wedge en(b) \wedge \neg en(m)$$

$$vbls_9 \equiv \neg en(a) \wedge \neg en(b) \wedge \neg en(m)$$

$$
\begin{aligned}
vbls_{10} \equiv\ & (|MUTEX1| = 0 \rightarrow en(b)) \wedge \\
& (|MUTEX1| = 1 \rightarrow \neg en(b)) \wedge \neg en(a) \wedge \neg en(m))
\end{aligned}
$$

$$
\begin{aligned}
vbls_{11} \equiv\ & (|MUTEX1| = 0 \rightarrow en(b)) \wedge \\
& (|MUTEX1| = 1 \rightarrow \neg en(b)) \wedge \neg en(a) \wedge en(m))
\end{aligned}
$$

$$
\begin{aligned}
vbls_{12} \equiv\ & (|MUTEX1| = 0 \rightarrow en(b)) \wedge \\
& (|MUTEX1| = 1 \rightarrow \neg en(b)) \wedge \neg en(a) \wedge \neg en(m))
\end{aligned}
$$

$$
\begin{aligned}
vbls_{13} \equiv\ & (|MUTEX1| = 0 \rightarrow en(b)) \wedge \\
& (|MUTEX1| = 1 \rightarrow \neg en(b)) \wedge \neg en(a) \wedge \neg en(m)) \wedge \\
& (|LP4| > 0 \rightarrow |WAITM| > 0) \wedge (|LP5| > 0 \rightarrow |WAITM| = 0)
\end{aligned}
$$

Figure 5.4 - The Expressions $vbls_i$

Before a proof by the induction principle will go through, it is necessary to further strengthen the invariant by conjoining the predicate [*aux*], where $aux \equiv aux_1 \wedge aux_2 \wedge \ldots \wedge aux_{21}$, and the definitions of the aux_i are given in Figure 5.5. The predicates $[aux_1]$, ... , $[aux_3]$, and $[aux_{10}]$, ... , $[aux_{13}]$ express relationships that are true in any system that uses blocked-set semaphores. The predicates $[aux_4]$, ... , $[aux_9]$ state that no execution sequence may contain two consecutive **P** operations on the same semaphore without an intervening **V** operation. This is why the general semaphores may be replaced with binary semaphores without any detrimental effect. The remainder of the $[aux_i]$ express some mutual exclusion conditions that are enforced by the system, and also relate the values of the variables *count1* and *count2* to the positions of tokens.

The "two-phase" nature of the execution of the system is reflected in the definitions of the thirteen categories. During phase one, the system state satisfies one of $[conf_1]$, ... , $[conf_{10}]$. System states during phase two are characterized by the predicates $[conf_{11}]$, ... , $[conf_{13}]$. Before the system state may change from phase one to phase two, the system must enter a state satisfying $[conf_{10}]$. Transitions between states in the various categories are made in such a way that a state satisfying $[conf_{10}]$ can only be entered by passing through a state satisfying $[conf_8]$ in which there were no processes blocked at the first **P**(*b*) in the trying region. This is the key property that makes the system starvation-free.

> **Claim 5.4** - The predicate [*inv*] is invariant for **Σ**, where $inv \equiv \vee_{i=1}^{13} (conf_i \wedge vbls_i) \wedge aux$.

Proof - by the induction principle is given in the Appendix. All the intuition about the system **Σ** has gone into the construction of the predicate [*inv*], what remains is simply tedious verification that could be done by machine. If one has a good intuitive understanding of the system **Σ**, the invariance of [*inv*] should seem plausible

$$
\begin{aligned}
aux_1 &\equiv a > 0 \rightarrow (a\_en = \varnothing \wedge a\_bl = \varnothing) \\
aux_2 &\equiv b > 0 \rightarrow (b\_en = \varnothing \wedge b\_bl = \varnothing) \\
aux_3 &\equiv m > 0 \rightarrow (m\_en = \varnothing \wedge m\_bl = \varnothing) \\
aux_4 &\equiv a \leq 1 \\
aux_5 &\equiv b \leq 1 \\
aux_6 &\equiv m \leq 1 \\
aux_7 &\equiv card(a\_en) \leq 1 \\
aux_8 &\equiv card(b\_en) \leq 1 \\
aux_9 &\equiv card(m\_en) \leq 1 \\
aux_{10} &\equiv \forall i(i \text{ at } TP7 \leftrightarrow (i \in a\_bl \vee i \in a\_en)) \\
aux_{11} &\equiv \forall i(i \text{ at } TP2 \cup TP9 \leftrightarrow (i \in b\_bl \vee i \in b\_en)) \\
aux_{12} &\equiv \forall i(i \text{ at } TP20 \leftrightarrow (i \in m\_bl \vee i \in m\_en)) \\
aux_{13} &\equiv \neg\exists i(i \in a\_bl \wedge i \in a\_en) \\
aux_{14} &\equiv \neg\exists i(i \in b\_bl \wedge i \in b\_en) \\
aux_{15} &\equiv \neg\exists i(i \in m\_bl \wedge i \in m\_en) \\
aux_{16} &\equiv |WAITB2| \leq 1 \\
aux_{17} &\equiv |MUTEX1| \leq 1 \\
aux_{18} &\equiv count1 = |TP5| + |TP6| + |TP7| + |TP8| + |TP9| + |TP10| + |TP11| \\
aux_{19} &\equiv \forall i(i \text{ at } TP4 \cup TP11 \rightarrow loc\_i = count1) \\
aux_{20} &\equiv count2 = |TP14| + |TP15| + |TP16| + |TP17| + |TP18| + |TP19| + \\
&\qquad |TP20| + |CR| + |LP1| + |LP2| \\
aux_{21} &\equiv \forall i(i \text{ at } TP13 \cup LP2 \rightarrow loc\_i = count2)
\end{aligned}
$$

Figure 5.5 - Auxiliary Invariants

at this point without the proof in the Appendix. ▩

We are now in a position to prove that Σ is deadlock-free.

> **Claim 5.5** - If q is a reachable state for Σ, such that no process is in the critical region in state q, but at least one process is in the trying or leaving region in state q, then there is a transition enabled in state q for some process in the trying or leaving region.

Proof - Since q is a reachable state for Σ, by Claim 5.4 we know that $[inv](q)$ is true. We split the proof into cases, depending upon which one of the predicates $[conf_i \wedge vbls_i]$ is true in state q. Since we will be concerned in the sequel only with the single given state q, no confusion can result if we omit the square-bracket notation. Thus we will write "$conf_1$ is true" to mean "$[conf_1](q)$ is true".

Case 1: $conf_1 \wedge vbls_1$ is true. Now, if there are any processes anywhere in the trying or leaving regions except at *TP1*, *WAITB1*, *WAITB2*, *WAITA*, or *WAITM*, then we are done, since those processes always have enabled transitions. Otherwise, $vbls_1$ implies that $en(b)$ is true, meaning that either $b > 0$ or $b\_en \neq \varnothing$.

Subcase 1a: If $b\_en \neq \varnothing$, then by aux_{11} we know that $\exists i(i \in b\_en \wedge i \text{ at } TP2 \cup TP9)$. This in turn implies that a process at *TP2* or *TP9* is enabled.

Subcase 1b: If $b > 0$, then by aux_2 we know that $b\_en = b\_bl = \varnothing$, and hence $|TP2 \cup TP9| = 0$ by aux_{11}. If there is a process at either *TP1* or *TP8* then that process is enabled, otherwise $|TP1 \cup WAITB1 \cup WAITB2| = 0$. Since we assume that there is at least one process in the trying or leaving region in state q, this means that there must be a process at either *WAITA* or *WAITM*. Since it follows from $vbls_1$ that $WAITM > 0$ implies $WAITA > 0$, we conclude that $WAITA > 0$.

Since $|WAITB2| = 0$, we know that $en(a)$ is true by $vbls_1$. Therefore, either $a > 0$ or $a\_en \neq \emptyset$. If $a > 0$, then by aux_1 and aux_{10}, there is a process at *TP6* and that process is enabled. If $a\_en \neq \emptyset$, $|\mathrm{TP7}| > 0$ by aux_{10}, and therefore a process at *TP7* is enabled.

Case 2: $conf_2 \wedge vbls_2$ is true. By $conf_2$, $|MUTEX1| > 0$. A process at *MUTEX1* always has an enabled transition.

Case 3: $conf_3 \wedge vbls_3$ is true. By $conf_3$, $|MUTEX2| > 0$. A process at *MUTEX2* always has an enabled transition.

Case 4: $conf_4 \wedge vbls_4$ is true. By $conf_4$, $|TP16| = 1$. A process at *TP16* always has an enabled transition.

Case 5: $conf_5 \wedge vbls_5$ is true. By $conf_5$, $|TP16| = 1$. A process at *TP16* always has an enabled transition.

Case 6: $conf_6 \wedge vbls_6$ is true. By $conf_6$, $|MUTEX1| = 1$. A process at *MUTEX1* always has an enabled transition.

Case 7: $conf_7 \wedge vbls_7$ is true. By $vbls_7$, $en(a)$ is true, which in turn implies that either $a > 0$ or $a\_en \neq \emptyset$.

Subcase 7a: If $a > 0$ then $a\_en = \emptyset \wedge a\_bl = \emptyset$ by aux_1, and $conf_7$ implies $|WAITA| = 1$. By aux_{10} we have that $|TP6| = 1$, and hence a process at *TP6* is enabled.

Subcase 7b: If $a\_en \neq \emptyset$ then $TP7 > 0$ by aux_{10} and hence a process at *TP7* is enabled.

Case 8: $conf_7 \wedge vbls_7$ is true. By $vbls_7$, $en(b)$ is true, which in turn implies that either $b > 0$ or $b\_en \neq \emptyset$.

Subcase 8a: If $b > 0$ then $b\_en = \emptyset \wedge b\_bl = \emptyset$ by aux_2, and $conf_8$ implies $|WAITB2| = 1$. By aux_{11} we have that $|TP8| = 1$, and hence a process at *TP8* is enabled.

Subcase 8b: If $b\_en \neq \emptyset$ then $|TP2| + |TP9| > 0$ by aux_{11}, and hence a process at *TP2* or *TP9* is enabled.

Case 9: $conf_9 \wedge vbls_9$ is true. By $conf_9$ we know that $|MUTEX2| = 1$, and a process at *MUTEX2* always has an enabled transition.

Case 10: $conf_{10} \wedge vbls_{10}$ is true. By $conf_{10}$ we know that $|TP18| = 1$, and a process at *TP18* is always enabled.

Case 11: $conf_{11} \wedge vbls_{11}$ is true. By $vbls_{11}$ we know that $en(m)$ is true, which in turn implies that either $m > 0$ or $m\_en \neq \emptyset$.

Subcase 11a: If $m > 0$ then $m\_en = \emptyset$ and $m\_bl = \emptyset$ by aux_3, and $conf_{11}$ implies $|WAITM| > 0$. By aux_{12} we have that $|TP19| > 0$, and hence a process at *TP19* is enabled.

Subcase 11b: If $m\_bl \neq \emptyset$ then $|TP20| > 0$ by aux_{12}, and hence a process at *TP20* is enabled.

Case 12: $conf_{12} \wedge vbls_{12}$ is true. This case does not arise, since the truth of $conf_{12}$ is in contradiction with our assumption that no process is in the critical region in state q.

Case 13: $conf_{13} \wedge vbls_{13}$ is true. By $conf_{13}$, we know that $|LR| > 0$, and a process at *LR* always has an enabled transition.

Since any reachable state must fall under one of Cases 1-13, the Claim is proved. ▩

Claim 5.6 - The system Σ is deadlock-free.

Proof - Let q be an arbitrary reachable state for Σ. We will show that Σ is deadlock-free by constructing an execution sequence α from q, such that there are no processes in the trying or leaving region in state $nxt(q, \alpha)$.

Define $\alpha_0 = \Lambda$. Now suppose for some $i \geq 0$, that α_i is an execution sequence from q. If there are no processes in the trying or leaving regions in state $nxt(q, \alpha_i)$, then set $\alpha = \alpha_i$ and we are done. Otherwise, by Claim 5.5, there is a process in the trying or leaving region with an enabled transition t. Define $\alpha_{i+1} = \alpha_i t_i$. This construction eventually terminates, since if it did not, that would imply the existence of an execution sequence in which some process fires infinitely many transitions without exiting the trying and leaving regions; a contradiction with the fact that the trying and leaving regions in Σ are acyclic. ▩

5.6 Freedom From Starvation

At this point, we could use the well-founded set method directly on the system Σ to prove that there are no execution sequences in which starvation occurs. However if we were to perform the proof in this way, we would quickly encounter difficulty due to the large number of details which would obscure the important points of the proof. Examples of such details are the exact values of variables and semaphores. The property of freedom from starvation is a statement about the movement of tokens during an execution sequence; the exact values of semaphores and variables are only important inasmuch as they serve to determine the movement of the tokens. It would be better if we could suppress such detail from our proof. The notion of a *parallel program homomorphism* serves this purpose.

5.6.1 Parallel Program Homomorphisms

A parallel program homomorphism from a parallel program Γ to a second parallel program Γ' is a mapping from the state set of Γ to the state set of Γ'. This mapping has the property that initial execution sequences are preserved: for any initial execution sequence α of Γ, there is a corresponding initial execution sequence α' of Γ'. However, since the parallel program homomorphism in general maps many states of Γ into a single state of Γ', the sequence α' may be shorter than α. The usefulness of parallel program homomorphisms derives from the fact that they allow us to "abstract" away from irrelevant details of Γ, and focus our attention on Γ', in which the properties of interest are more clearly displayed.

Definition 5.7 - Let Γ and Γ' be parallel programs. A mapping h from $\mathcal{Q}_\Gamma$ to $\mathcal{Q}_{\Gamma'}$ is a *parallel program homomorphism* if the following hold:

(1) If q_I is the initial state for Γ, and q'_I is the initial state for Γ', then $h(q_I) = q'_I$.

(2) For any reachable state q and transition t for Γ enabled in state q, if $h(nxt(q, t)) \neq h(q)$, then there exists an unique transition t' in Γ' such that $h(nxt(q, t)) = nxt(h(q), t')$.

Definition 5.8 - Let h be a parallel program homomorphism from Γ to Γ'. Define the mapping $image_h$ from initial execution sequences of Γ to initial execution sequences of Γ' as follows:

(1) $image_h(\Lambda) = \Lambda$

(2) Suppose α is an initial execution sequence for Γ, $q = nxt(q_I, \alpha)$, and t is a transition enabled in state q. Then:

(a) If $h(nxt(q, t)) = h(q)$, then $image_h(\alpha t) = image_h(\alpha)$.

(b) If $h(nxt(q, t)) \neq h(q)$, then $image_h(\alpha t) = image_h(\alpha)t'$, where t' is the unique transition of Γ' such that $h(nxt(q, t)) = nxt(h(q), t')$.

Suppose h is a parallel program homomorphism from Γ to Γ'. Suppose P and R are predicates on execution sequences of Γ and Γ', respectively, such that if α is an initial execution sequence for Γ then $R(image_h(\alpha))$ implies $P(\alpha)$. The "parallel program homomorphism method" simply states that if we can show that every initial execution sequence for Γ' satisfies R, then we know that every initial execution sequence for Γ satisfies P. This is the idea we will use in proving that Σ is starvation-free.

Figure 5.6 shows the graph of a new parallel program Δ, which is a "condensed" version of Σ, in the sense that unimportant details of the execution of Σ are suppressed in Δ. Note that Δ is *not* a system of processes. After defining a parallel program homomorphism h from Σ to Δ, we will be able to replace proofs of properties of Σ by simpler proofs of corresponding properties of Δ. Another benefit of this approach is that a clean separation may be made between the intuitive parts of the proof, presented in this section, and the mechanical details, which are left to the Appendix.

The variables of Δ are *ncr*, *tp1*, *waitb1*, *mutex1*, *waita*, *waitb2*, *mutex2*, *tp16*, *tp18*, *waitm*, *cr*, *lr*, and *pnc*. The places of Δ are CP1, CP2, CP3, CP4, CP5, CP6, CP7, and CP8. The initial state r_I for Δ (we will use the letter "r" to denote states, and "u" to denote transitions of Δ) assigns the value one to places CP1 and CP8, the value N to the variable *ncr*, and the value zero to all other places and variables. Each horizontal bar in Figure 5.6 stands for one or more transitions of Δ, and is labeled to correspond with the list of statement labels in Figure 5.7. For example, the horizontal bar labeled "1a" in Figure 5.6 actually stands for seven distinct transitions, whose statement labels are "1a-1" to "1a-7" in Figure 5.7.

Figure 5.8 defines a mapping h from states of Σ to states of Δ. Note that if q is a state of Σ, then the values of the variables *ncr*, *tp1*, ... of Δ in state $h(q)$ equal the number of processes whose tokens are at places in *NCR*, *TP1*, ... , respectively, in Σ. This reflects the idea that only the configuration of tokens is important in proving freedom from starvation; the exact values of variables and semaphores are unimportant.

In any reachable state of Δ, there are two tokens on the graph. One token is always on place CP8. The firing of transitions 8a-1 or 8a-2 models a process in Σ leaving the noncritical region and entering the trying region. The second token in Δ is always on one of places CP1-CP7 in Δ. The position of this token reflects which of the $conf_i$ is true in Σ.

Claim 5.9 - The mapping h is a parallel program homomorphism from Σ to Δ.

Proof - is in the Appendix. ▩

It is important, if we are to use Δ to discuss the starvation properties of Σ, that the mapping $image_h$ preserve infinite execution sequences. This is the content of the next Claim.

Claim 5.10 - If α is an infinite initial execution sequence for Σ, then $image_h(\alpha)$ is also infinite.

Proof - Suppose $\alpha = t_0 t_1 \ldots$ is an infinite initial execution sequence for Σ, with corresponding state sequence $q_0 q_1 \ldots$. Then, since there are only a finite number of processes in Σ, there must be an i such that process i has infinitely many transitions

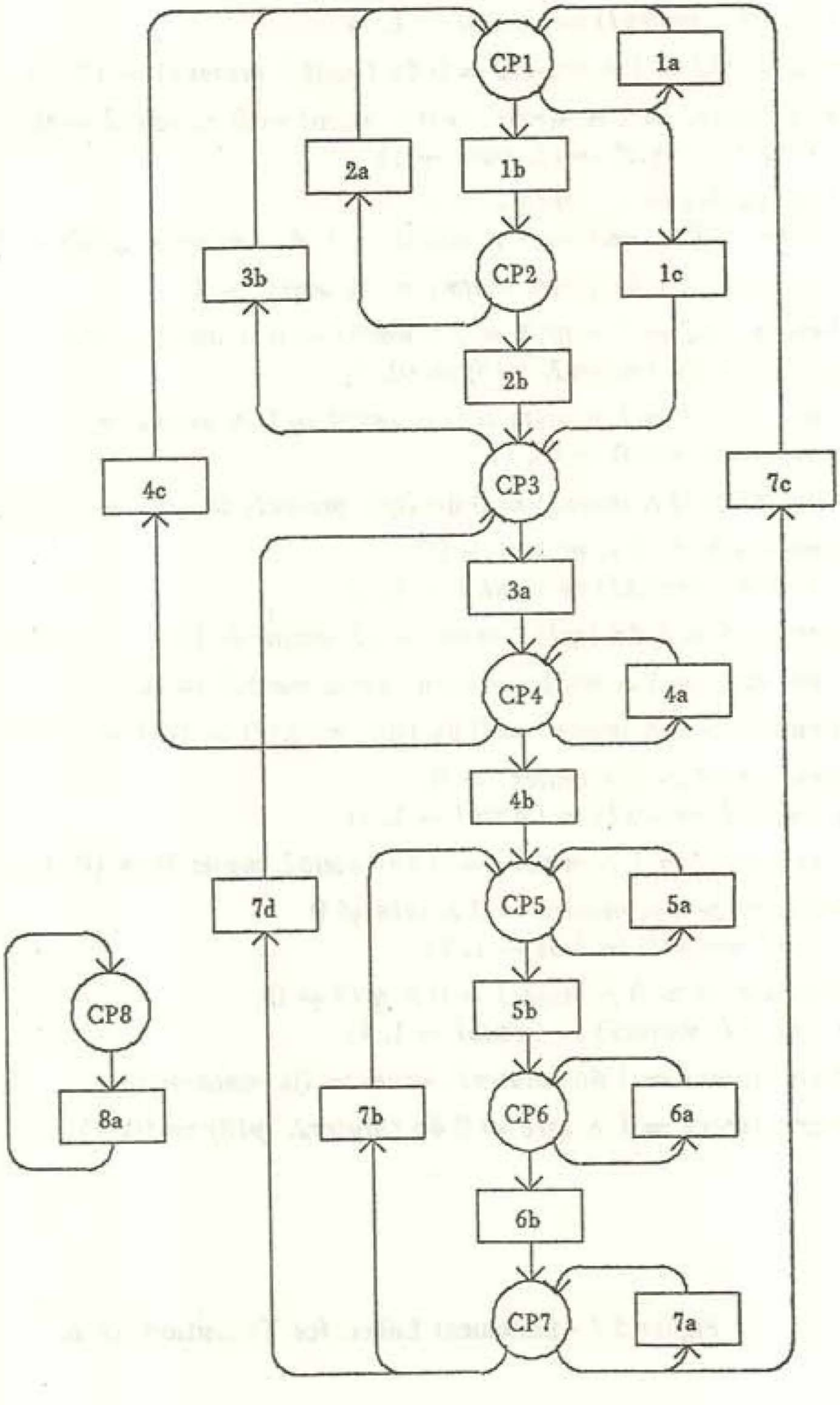


Figure 5.6 - The Parallel Program Δ

1a-1: **when** $waita > 0 \wedge waitb2 = 0$ **do** $(waita, waitb2) := (waita - 1, 1)$

1a-2: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

1a-3: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

1a-4: **when** $waitb2 = 1 \wedge mutex2 = 0$ **do** $(waitb2, mutex2) := (0, 1)$

1a-5: **when** $mutex1 = 1 \wedge \neg(waita = 0 \wedge waitb1 = 0 \wedge waitb2 = 0)$
do $(mutex1, waita) := (0, waita + 1)$

1a-6: **when** $mutex2 = 1 \wedge tp16 = 0 \wedge$
$\neg(waitb1 = 0 \wedge waita = 1 \wedge waitb2 = 0)$ **do** $(mutex2, tp16) := (0, 1)$

1a-7: **when** $tp16 = 1$ **do** $(tp16, waitm) := (0, waitm + 1)$

1b: **when** $mutex2 = 1 \wedge tp16 = 0 \wedge waitb1 = 0 \wedge waita = 1 \wedge$
$waitb2 = 0$ **do** $(mutex2, tp16) := (0, 1)$

1c: **when** $mutex1 = 1 \wedge waita = 0 \wedge waitb1 = 0 \wedge waitb2 = 0$
do $(mutex1, waita) := (0, 1)$

2a-1: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

2a-2: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

2b: **when** $tp16 = 1$ **do** $(tp16, waitm) := (0, waitm + 1)$

3a: **when** $waita = 1 \wedge waitb2 = 0$ **do** $(waita, waitb2) := (0, 1)$

3b-1: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

3b-2: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

4a-1: **when** $waitb2 = 1 \wedge mutex2 = 0$ **do** $(waitb2, mutex2) := (0, 1)$

4a-2: **when** $tp1 > 0 \wedge mutex1 = 0 \wedge tp18 \neq 0$
do $(tp1, mutex1) := (tp1 - 1, 1)$

4a-3: **when** $waitb1 > 0 \wedge mutex1 = 0 \wedge tp18 \neq 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

4a-4: **when** $mutex1 = 1$ **do** $(mutex1, waita) := (0, waita + 1)$

4a-5: **when** $mutex2 = 1 \wedge tp18 = 0$ **do** $(mutex2, tp18) := (0, 1)$

Figure 5.7 - Statement Labels for Transitions of Δ

4b: **when** $tp18 = 1$ **do** $(tp18, waitm) := (0, waitm + 1)$

4c-1: **when** $tp1 > 0 \wedge mutex1 = 0 \wedge tp18 = 0$
do $(tp1, mutex1) := (tp1 - 1, 1)$

4c-2: **when** $waitb1 > 0 \wedge mutex1 = 0 \wedge tp18 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

5a-1: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

5a-2: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

5a-3: **when** $mutex1 = 1$ **do** $(mutex1, waita) := (0, waita + 1)$

5b: **when** $waitm > 0 \wedge cr = 0$
do $(waitm, cr) := (waitm - 1, 1)$

6a-1: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

6a-2: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1) := (waitb1 - 1, 1)$

6a-3: **when** $mutex1 = 1$ **do** $(mutex1, waita) := (0, waita + 1)$

6b: **when** $cr = 1 \wedge lr = 0$ **do** $(cr, lr) := (0, 1)$

7a-1: **when** $tp1 > 0 \wedge mutex1 = 0$ **do** $(tp1, mutex1) := (tp1 - 1, 1)$

7a-2: **when** $waitb1 > 0 \wedge mutex1 = 0$
do $(waitb1, mutex1, \mathrm{pnc}) := (waitb1 - 1, 1, 0)$

7a-3: **when** $mutex1 = 1 \wedge waitb1 = 0$ **do** $(mutex1, waita) := (0, waita + 1)$

7a-4: **when** $mutex1 = 1 \wedge waitb1 > 0$
do $(mutex1, waita, pnc) := (0, waita + 1, 1)$

7b: **when** $lr = 1 \wedge waitm > 0$ **do** $(lr, ncr) := (0, ncr + 1)$

7c: **when** $lr = 1 \wedge waitm = 0 \wedge (mutex1 > 0 \vee (waita = 1 \rightarrow pnc = 1))$
do $(lr, ncr) := (0, ncr + 1)$

7d: **when** $lr = 1 \wedge waitm = 0 \wedge mutex1 = 0 \wedge waita = 1 \wedge pnc = 0$
do $(lr, ncr) := (0, ncr + 1)$

8a-1: **when** $ncr > 0$ **do** $(ncr, tp1) := (ncr - 1, tp1 + 1)$

8a-2: **when** $tp1 > 0$ **do** $(tp1, waitb1) := (tp1 - 1, waitb1 + 1)$

Figure 5.7 - Statement Labels for Transitions of Δ (cont.)

$$(h(q))(ncr) = [\![|NCR| + |LP6| + |LP7|]\!](q)$$
$$(h(q))(tp1) = [\![TP1]\!](q)$$
$$(h(q))(waitb1) = [\![WAITB1]\!](q)$$
$$(h(q))(mutex1) = [\![MUTEX1]\!](q)$$
$$(h(q))(waita) = [\![WAITA]\!](q)$$
$$(h(q))(waitb2) = [\![WAITB2]\!](q)$$
$$(h(q))(mutex2) = [\![MUTEX2]\!](q)$$
$$(h(q))(tp16) = [\![TP16]\!](q)$$
$$(h(q))(tp18) = [\![TP18]\!](q)$$
$$(h(q))(waitm) = [\![WAITM]\!](q)$$
$$(h(q))(cr) = [\![|CR| + |TP21|]\!](q)$$
$$(h(q))(lr) = [\![LR]\!](q)$$
$$(h(q))(pnc) = \textbf{if } [conf_{13} \wedge procnotcounted](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP1}) = \textbf{if } [conf_1 \vee conf_2 \vee conf_3 \vee conf_4 \vee conf_6](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP2}) = \textbf{if } [conf_5](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP3}) = \textbf{if } [conf_7](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP4}) = \textbf{if } [conf_8 \vee conf_9 \vee conf_{10}](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP5}) = \textbf{if } [conf_{11}](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP6}) = \textbf{if } [conf_{12}](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP7}) = \textbf{if } [conf_{13}](q) \textbf{ then } 1 \textbf{ else } 0$$
$$(h(q))(\mathrm{CP8}) = 1$$

Figure 5.8 - The Mapping h From $\mathfrak{Q}_\Sigma$ to $\mathfrak{Q}_\Delta$

in α. Since the trying and leaving regions in the graph for process i are acyclic, process i must pass from the noncritical region to the trying region infinitely often in α. There must therefore be an infinite sequence $0 \leq k_1 < k_2 < \ldots$, such that for all $j \geq 1$ process i is in the noncritical region in state q_{k_j}, and process i is in the trying region in state q_{k_j+1}. Define $\beta_j = seq(\alpha, 0, k_j)$. Since $(h(q_{k_j+1}))(ncr) = (h(q_{k_j}))(ncr) - 1 \neq (h(q_{k_j}))(ncr)$, there must be a uniquely determined transition u_j of Δ such that $image_h(\beta_j t_{k_j}) = image_h(\beta_j)u_j$. Since there are infinitely many numbers k_j, there are infinitely many corresponding u_j in $image_h(\alpha)$, and hence $image_h(\alpha)$ is infinite. ▩

5.6.2 Proof of Freedom From Starvation

We will now prove several properties (Claims 5.11-5.18) of the parallel program Δ. Each of these properties implies the truth of a corresponding property of Σ, however the properties are much easier to state and verify for Δ. Claims 5.11 and 5.12 describe some simple invariants on the state of Δ. Claims 5.13 and 5.14 are statements about Δ that imply the "two-phase" nature of the execution of Σ. Claims 5.15-5.18 show that in any infinite execution sequence for Σ, no process can wait forever at any of the **P** operations in Σ. In view of the finite delay property and the fact that the trying and leaving regions in Σ are acyclic, this implies that Σ is starvation-free.

> **Claim 5.11** - [CP1 + CP2 + CP3 + CP4 + CP5 + CP6 + CP7 = 1] is invariant for Δ.

Proof - by the induction principle. Note that (r_1)(CP1) = 1, and (r_1)(CPi) = 0 for $2 \leq i \leq 7$. It is easy to verify that no transition of Δ changes the total number of tokens on places CP1-CP7. ▩

Claim 5.12 - $[ncr + tp1 + waitb1 + waita + waitb2 + mutex2 + tp16 + tp18 + waitm + cr + lr = N]$ is invariant for Δ.

Proof - by the induction principle is easy. ▩

Claim 5.13 - Let $\alpha = u_0 u_1 \ldots$ be an infinite initial execution sequence for Δ, with corresponding state sequence $r_0 r_1 \ldots$. Let $P_1(r)$ be the predicate $[\mathrm{CP1} + \mathrm{CP2} + \mathrm{CP3} + \mathrm{CP4} = 1]$, and let $P_2(r)$ be the predicate $[\mathrm{CP5} + \mathrm{CP6} + \mathrm{CP7} = 1]$. Then for any $k \geq 0$:

(a) there exists an $a \geq k$ such that $P_1(r_a)$ is true; and

(b) there exists an $a' \geq k$ such that $P_2(r_{a'})$ is true.

Proof - of both statements (a) and (b) is by the well-founded set method.

(a) If $P_1(r_k)$ is true then let $a = k$. Otherwise, by Claim 5.11 we know that $P_2(r_k)$ is true. Let $(\mathbb{N}, <)$ be the well-founded set consisting of the natural numbers under the usual ordering, and define the function f from $\mathbb{Q}_\Delta$ to $\mathbb{N}$ by:

$$f(r) = [N(N+1)^7 - (waitm + (N+1)cr + (N+1)^2 lr + (N+1)^3 ncr + (N+1)^4 tp1 + (N+1)^5 waitb1 + (N+1)^6 mutex1 + (N+1)^7 waita)](r)$$

By Claim 5.12, $f(r) \geq 0$ for all reachable states r. By examination of the labels of the transitions of Δ, it is straightforward to verify that for any state r, and transition u for Δ enabled in state r, $P_2(r)$ and $P_2(nxt(r, u))$ imply that $f(nxt(r, u)) < f(r)$. By the well-founded set method, there must therefore exist $a \geq k$ such that $P_2(r_a)$ is false. But by Claim 5.12 this means that $P_1(r_a)$ is true. Thus (a) is proved.

(b) The argument for (b) is entirely analogous, with the roles of P_1 and P_2 interchanged, and the function f defined by:

$$f(r) = [N(N+1)^7 - (\mathit{tp1} + (N+1)\mathit{waitb1} + (N+1)^2\mathit{mutex1} + (N+1)^3\mathit{waita} + (N+1)^4\mathit{waitb2} + (N+1)^5\mathit{mutex2} + (N+1)^6(\mathit{tp16}+\mathit{tp18}) + (N+1)^7\mathit{waitm})](r) \;\blacksquare$$

Claim 5.14 below is a somewhat strengthened version of Claim 5.13.

> **Claim 5.14** - In the situation of Claim 5.13, for any $k \geq 0$:
>
> (a) there exists an $a \geq k$ such that $[\mathrm{CP1} + \mathrm{CP3}](r_a) = 1$; and
>
> (b) there exists an $a' \geq k$ such that $[\mathrm{CP5}](r_{a'}) = 1$.

Proof - (a) By Claim 5.13, we may select a and b, with $k \leq b < a$, such that $P_2(r_b)$ and $P_1(r_a)$ are true, where the predicates P_1 and P_2 are defined as in Claim 5.13. Since every path in the graph of Δ from places CP5-CP7 to places CP1-CP4 must pass through either place CP1 or CP3, we may choose a so that $[\mathrm{CP1} + \mathrm{CP3}](r_a)$ is true. (b) By Claim 5.13, we may select a' and b', with $k \leq b' < a'$, such that $P_1(r_{b'})$ and $P_2(r_{a'})$ are true. Since every path in the graph of Δ from places CP1-CP4 to places CP5-CP7 must pass through place CP5, we may choose a' such that $[\mathrm{CP5}](r_{a'}) = 1$. ■

> **Claim 5.15** - In the situation of Claim 5.13, for any k there is an $a \geq k$ such that $(r_a)(\mathit{waitb2}) = 0$.

Proof - We split the proof into cases, depending upon which of $(r_k)(\mathrm{CP1})$, ... , $(r_k)(\mathrm{CP7})$ equals one.

Case 1: $(r_k)(\mathrm{CP1}) = 1$. By Claim 5.14, there is a $b > k$ such that $(r_b)(\mathrm{CP5}) = 1$. Examination of the graph of Δ shows that for this to occur, u_a must be either the transition labeled "1b" or "1c" in Figure 5.6, for some number a, with $k \leq a < b$. But for either transition "1b" or "1c" to be enabled in state r_a requires that

$(r_a)(waitb2) = 0$.

Case 2: $[\mathsf{CP2} + \mathsf{CP3}](r_k) = 1$. By Claim 5.14, there is a $b \geq k$ such that $(r_b)(\mathsf{CP5}) = 1$. Examination of the graph of Δ shows that for this to occur, u_a must be one of the transitions labeled "1b", "1c", or "3a", for some a, with $k \leq a < b$. But then for u_a to be enabled in state r_a requires $(r_a)(waitb2) = 0$.

Case 3: $[\mathsf{CP4} + \mathsf{CP5} + \mathsf{CP6} + \mathsf{CP7}](r_k) = 1$. By Claim 5.14, there exists a $k' > k$ such that $[\mathsf{CP1} + \mathsf{CP3}](r_{k'}) = 1$. But then either the argument of Case 1 or Case 2 applied with k' in place of k shows the existence of $a \geq k'$, with $(r_a)(waitb2) = 0$. ▩

Claim 5.16 - In the situation of Claim 5.13, for any k there is an $a \geq k$ such that $(r_a)(waita) = 0$.

Proof -

Case 1: $[\mathsf{CP1} + \mathsf{CP2} + \mathsf{CP3}](r_k) = 1$. By Claim 5.14, there exists $b > k$ such that $(r_b)(\mathsf{CP5}) = 1$. This can only happen if u_{a-1} is the transition labeled "3a" in Figure 5.6, for some number a, with $k < a < b$. But this means that $(r_a)(waita) = 0$.

Case 2: $[\mathsf{CP4} + \mathsf{CP5} + \mathsf{CP6} + \mathsf{CP7}](r_k) = 1$. By Claim 5.14, there exists $k' > k$ such that $[\mathsf{CP1} + \mathsf{CP3}](r_{k'}) = 1$. Application of the argument of Case 1 with $k = k'$ now shows the existence of $a \geq k'$ with $(r_a)(waita) = 0$. ▩

Claim 5.17 - In the situation of Claim 5.13, for any k there is an $a \geq k$ such that $(r_a)(waitm) = 0$.

Proof -

Case 1: $[\mathrm{CP5}+\mathrm{CP6}+\mathrm{CP7}](r_k)=1$. By Claim 5.14, there exists $b>k$ with $[\mathrm{CP1}+\mathrm{CP3}](r_b)=1$. But this means that u_a is either the transition labeled "7c" or "7d" in Figure 5.6, for some number *a*, with $k \leq a < b$. For either transition "7c" or "7d" to be enabled in state r_a it must be the case that $(r_a)(\mathit{waitm})=0$.

Case 2: $[\mathrm{CP1}+\mathrm{CP2}+\mathrm{CP3}+\mathrm{CP4}](r_k)=1$. By Claim 5.14, there exists $k'>k$ with $(r_{k'})(\mathrm{CP5})=1$. Application of the argument of Case 1 with *k* replaced by k' now shows the existence of $a \geq k'$ with $(r_a)(\mathit{waitm})=0$. ▩

Claim 5.18 - In the situation of Claim 5.13, for any *k* there is an $a \geq k$ such that $(r_a)(\mathit{waitbI})=0$.

Proof -

Case 1: $[\mathrm{CP1}](r_k)=1$. By Claim 5.14, there exists $b>k$ with $[\mathrm{CP5}](r_b)=1$. This means that u_a is either the transition labeled "1b" or "1c" in Figure 5.6, for some number *a*, with $k \leq a < b$. For either of these transitions to be enabled, we must have $(r_a)(\mathit{waitbI})=0$.

Case 2: $[\mathrm{CP5}+\mathrm{CP6}+\mathrm{CP7}](r_k)=1$. By Claim 5.14, there exists $k'>k$ with $[\mathrm{CP1}+\mathrm{CP3}](r_{k'})=1$. We may assume, without loss of generality, that k' is the least such number.

Subcase 2a: If $[\mathrm{CP3}](r_{k'})=1$, then there must be an *a*, with $k \leq a < k'$ such that u_a is the transition labeled "7d" in Figure 5.6. For transition "7d" to be enabled, it must be the case that $(r_a)(\mathit{waitbI})=0$.

Subcase 2b: If instead, $[\mathrm{CP1}](r_b)=1$, then application of the argument of Case

1, with k replaced by k', shows the existence of $a \geq k'$ with $(r_a)(\mathit{waitb1}) = 0$.

Case 3: $[\mathsf{CP2} + \mathsf{CP3} + \mathsf{CP4}](r_k) = 1$. By Claim 5.14, there exists $k'' > k$ with $[\mathsf{CP5}](r_{k''}) = 1$. Application of the argument of Case 2 with k replaced by k'' now shows the existence of $a \geq k''$ such that $(r_a)(\mathit{waitb1}) = 0$. ▩

Claim 5.19 - The mutual exclusion system Σ is starvation-free.

Proof - Suppose not. That is, suppose there is a valid infinite initial execution sequence α for Σ, with corresponding state sequence $q_0q_1 \ldots$, such that:

(1) There exists a process i such that for all $k \geq 0$, there is an $a \geq k$ and an $a' \geq k$ with process i in the critical region in state q_a, and process i not in the critical region in state $q_{a'}$; and

(2) There is a process i' and a $k' \geq 0$, such that for all $j \geq k'$ process i' is in either the trying or leaving region in state q_j.

Now α must contain only a finite number of transitions for process i'. If this were not true, then process i' could not be in the trying or leaving region for all $j \geq k'$, since these regions are acyclic. Since α is assumed valid, there must be some n such that for all $j \geq n$, process i' has no enabled transition in state q_j. This in turn implies that for all $j \geq n$, the token for process i' is at one of the four places $\mathsf{TP2}$, $\mathsf{TP7}$, $\mathsf{TP9}$, or $\mathsf{TP20}$ in state q_j.

Let $\beta = \mathit{image}_h(\alpha)$. Since α contains infinitely many transitions for processes in the trying and leaving regions, β is infinite by Claim 5.10. Suppose β has corresponding state sequence $r_0r_1 \ldots$. Then, by Claims 5.15-5.18, for any m there exist a_1, a_2, a_3, and $a_4 \geq m$ such that $(r_{a_1})(\mathit{waitb1}) = (r_{a_2})(\mathit{waitb2}) = (r_{a_3})(\mathit{waita}) = (r_{a_4})(\mathit{waitm}) = 0$. But by the definition of h, this implies the existence of a'_1, a'_2, a'_3, and $a'_4 \geq n$, with $[TP2](q_{a'_1}) = [TP7](q_{a'_2}) =$

$[TP9](q_{a_3'}) = [TP20](q_{a_4'}) = 0$. This is a contradiction, and we conclude that α cannot exist and that Σ is starvation-free. ■

5.7 Conclusion

That the system of Figure 5.9 is a solution to the starvation-free mutual exclusion problem is a direct consequence of Claims 5.1, 5.3, 5.6, and 5.19.

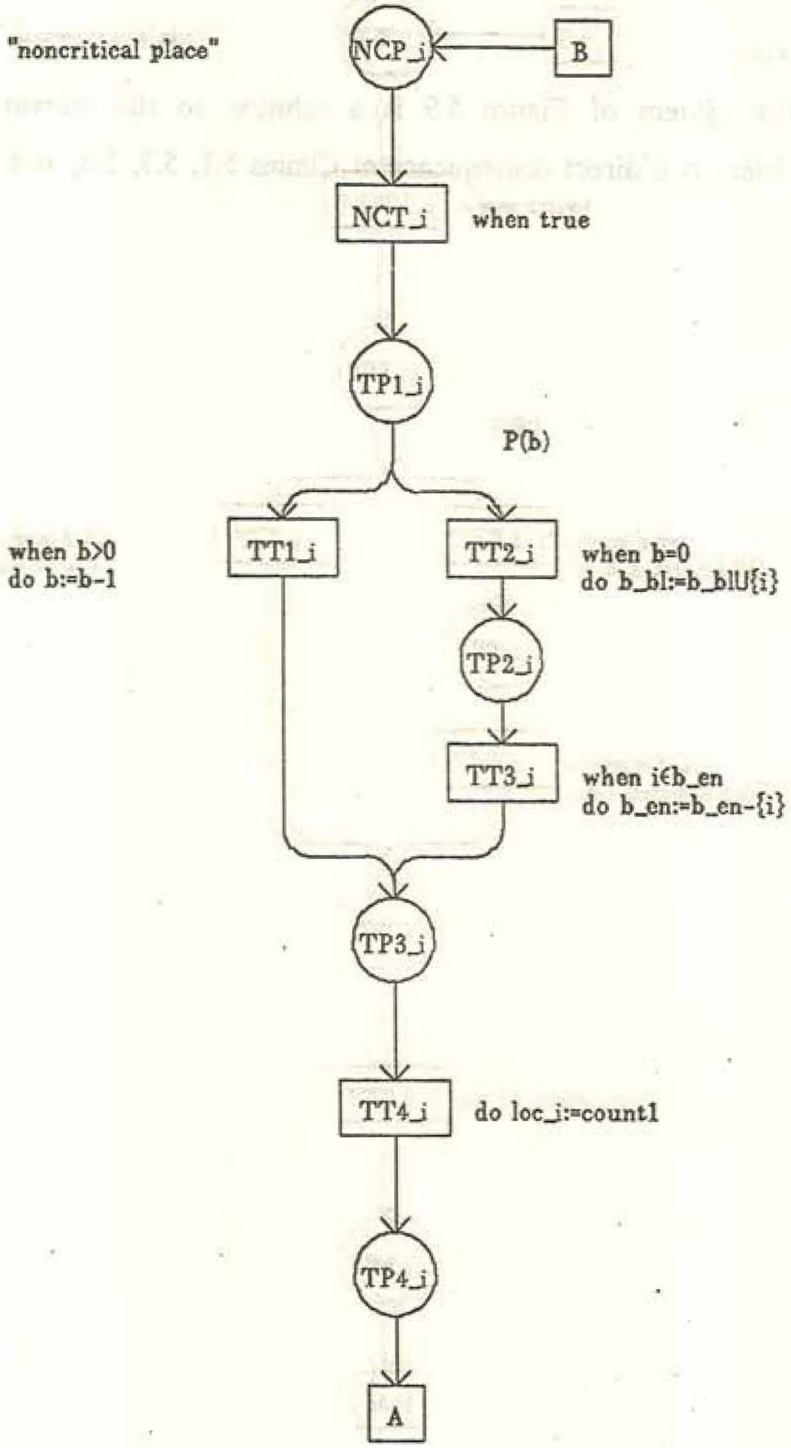


Figure 5.9(a) - Morris' Solution in Graphical Form

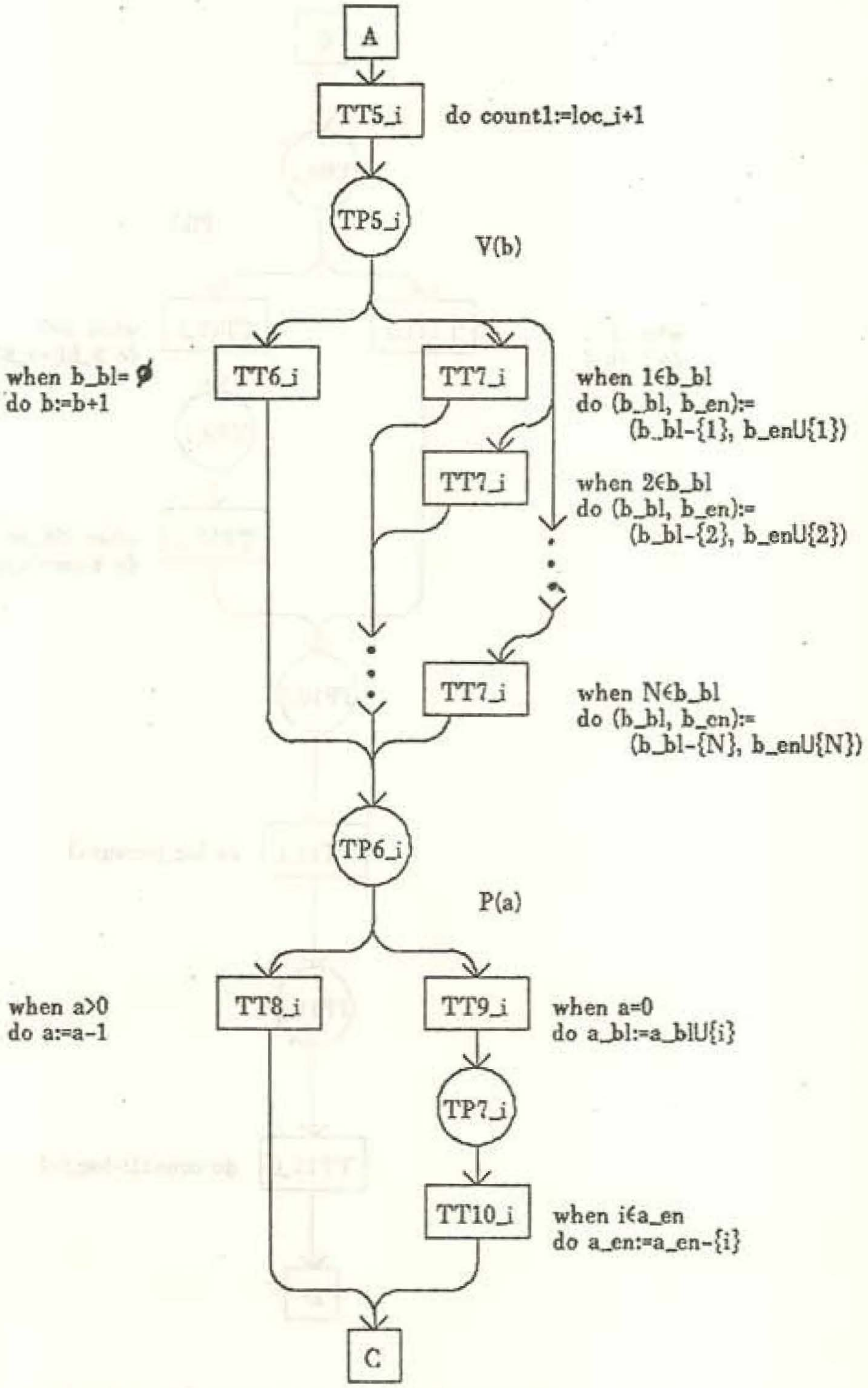


Figure 5.9(b) - Morris' Solution in Graphical Form

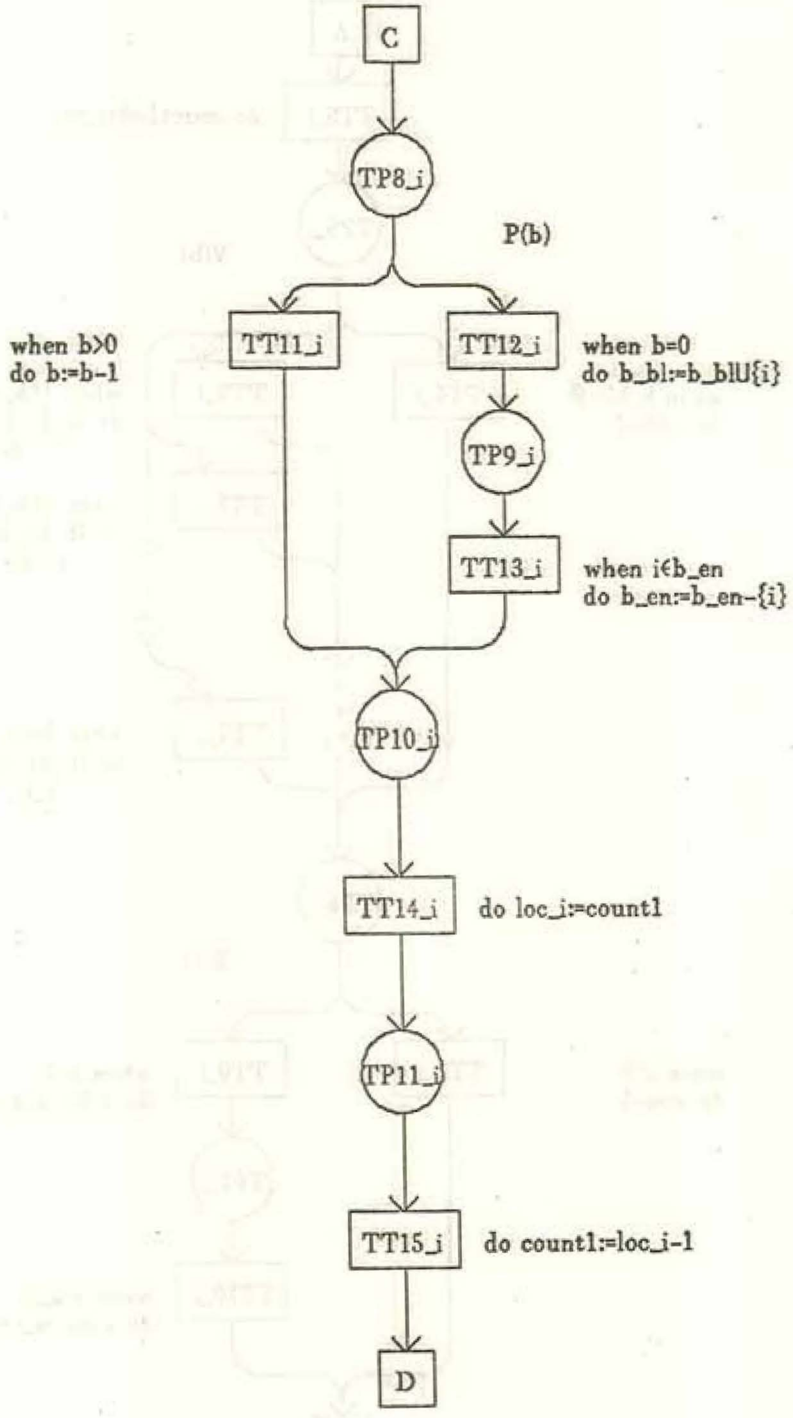


Figure 5.9(c) - Morris' Solution in Graphical Form

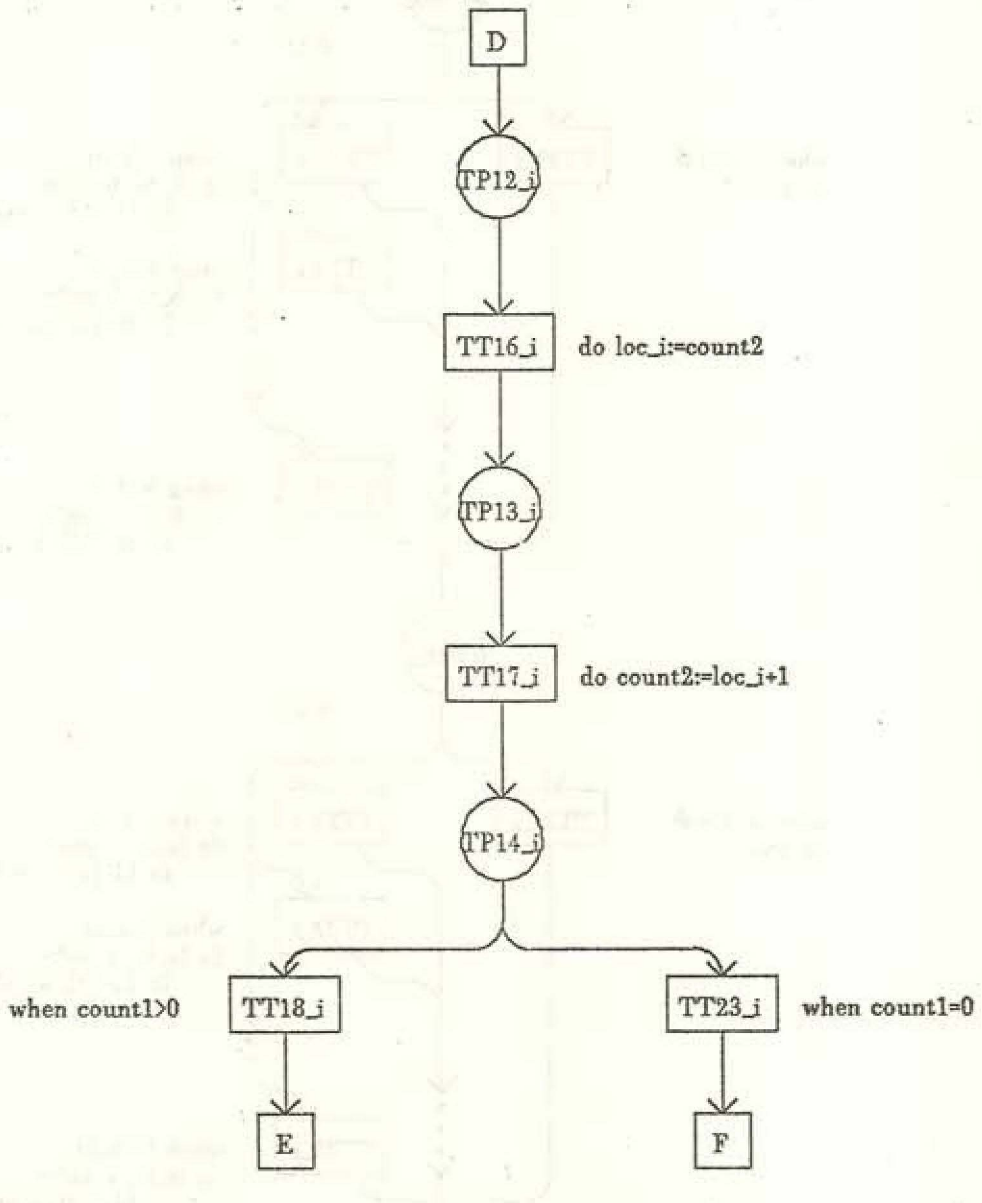


Figure 5.9(d) - Morris' Solution in Graphical Form

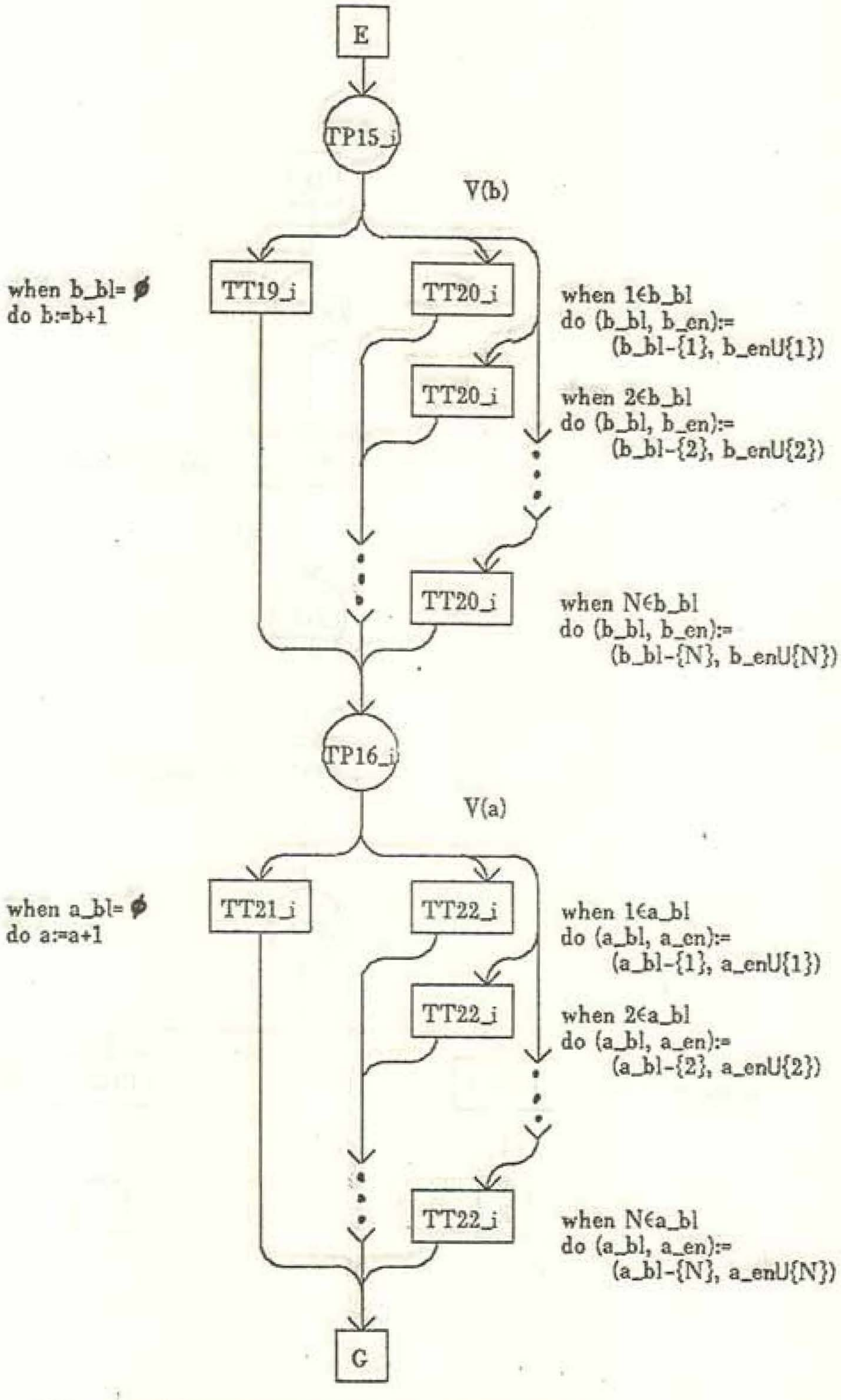


Figure 5.9(e) - Morris' Solution in Graphical Form

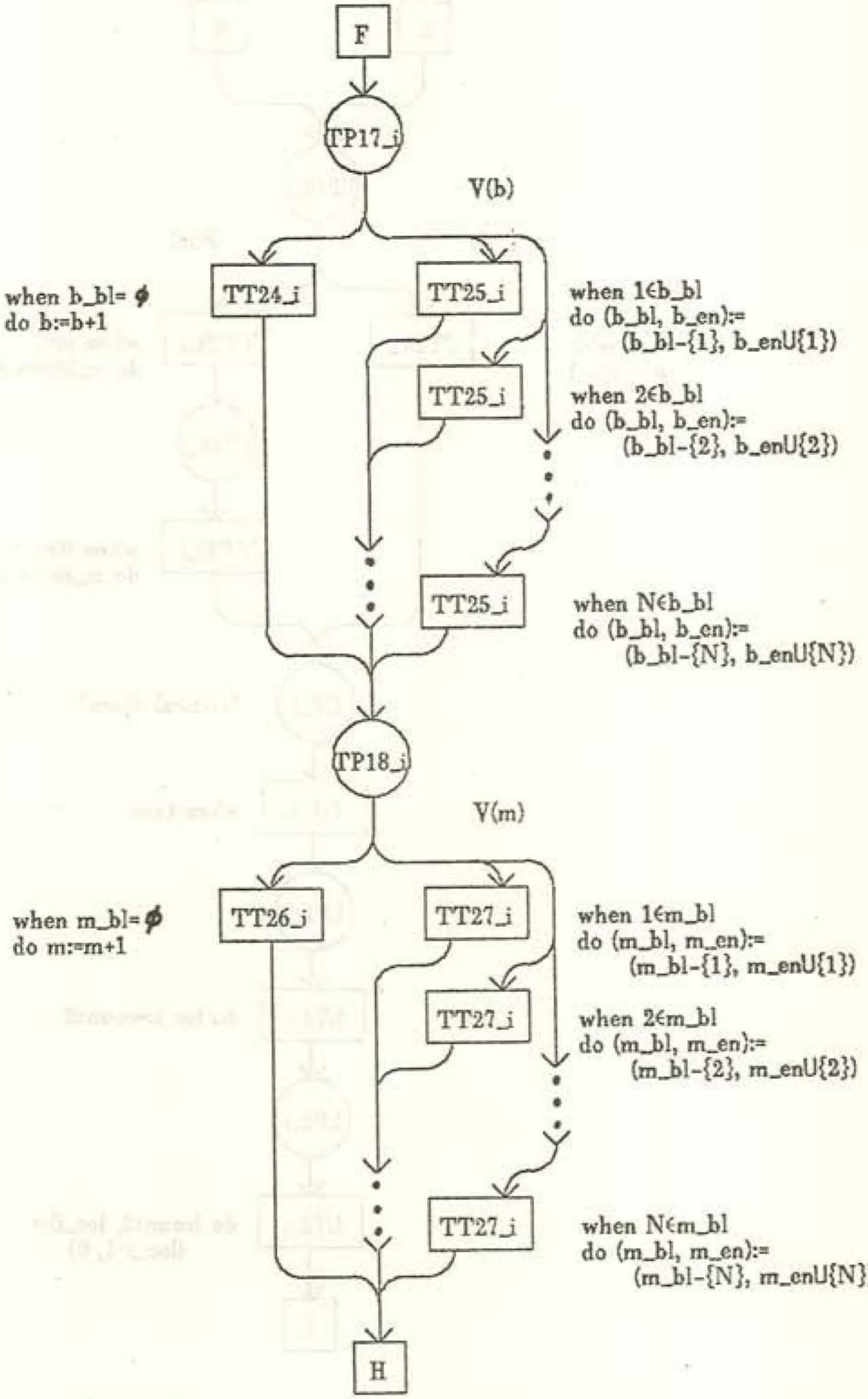


Figure 5.9(f) - Morris' Solution in Graphical Form

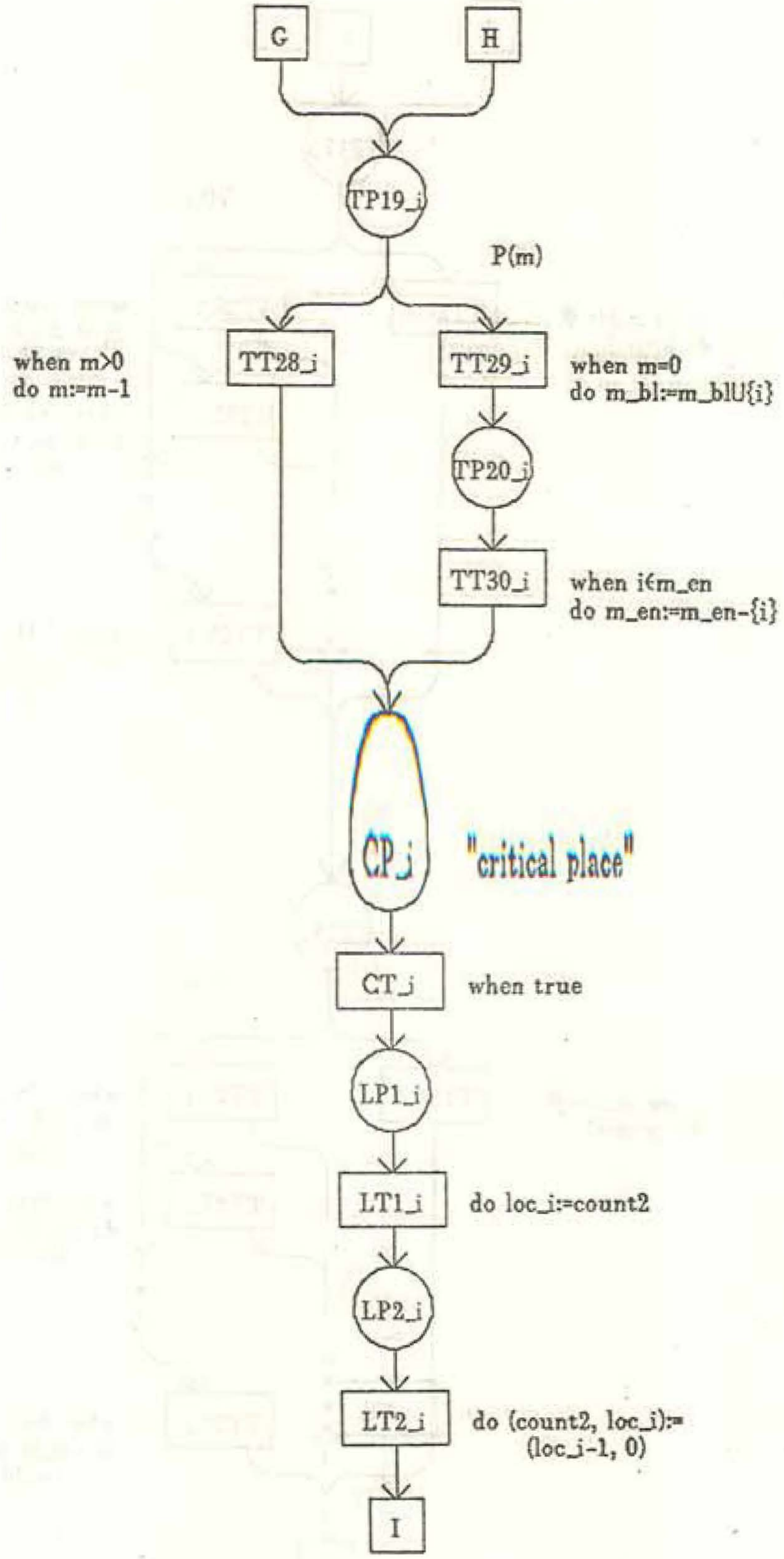


Figure 5.9(g) - Morris' Solution in Graphical Form

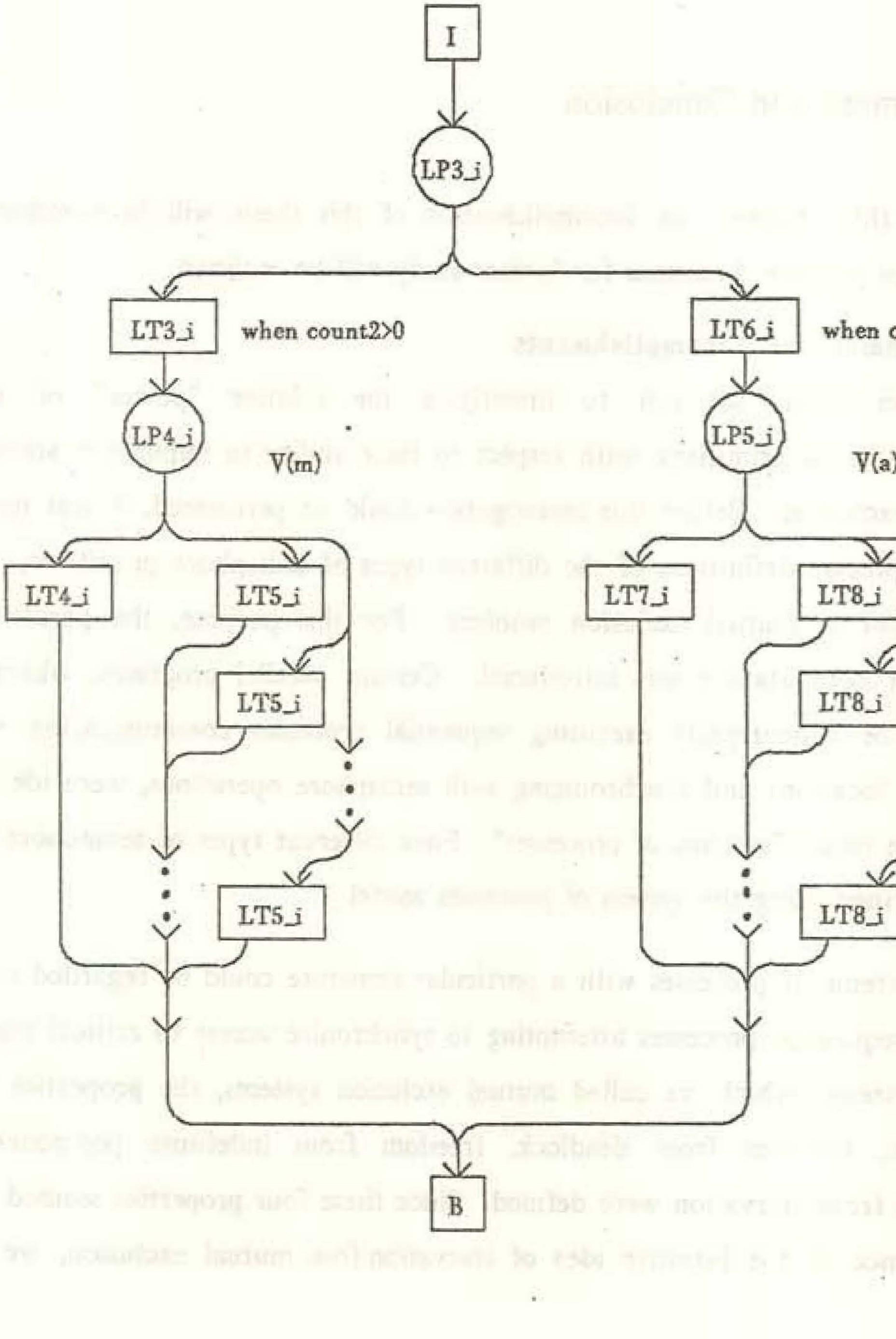


Figure 5.9(h) - Morris' Solution in Graphical Form

6. Summary and Conclusion

In this chapter, the accomplishments of this thesis will be summarized, and then some possible directions for further study will be outlined.

6.1 Summary of Accomplishments

This thesis set out to investigate the relative "power" of semaphore synchronization primitives, with respect to their ability to implement starvation-free mutual exclusion. Before this investigation could be performed, it was necessary to provide precise definitions of the different types of semaphore primitives, and of the starvation-free mutual exclusion problem. For this purpose, the parallel program model of computation was introduced. Certain parallel programs, which model a number of concurrently executing sequential processes communicating via shared memory locations and synchronizing with semaphore operations, were identified and given the name "systems of processes". Four different types of semaphore primitives were defined using the system of processes model.

Systems of processes with a particular structure could be regarded as modeling several sequential processes attempting to synchronize access to critical regions. For these systems, which we called mutual exclusion systems, the properties of mutual exclusion, freedom from deadlock, freedom from indefinite postponement, and freedom from starvation were defined. Since these four properties seemed to capture the essence of the intuitive idea of starvation-free mutual exclusion, we defined a solution to the starvation-free mutual exclusion problem to be a mutual exclusion system with these properties.

After completing the underlying definitions, it became possible to proceed with the formal investigation. The first discovery was that semaphore operations are not needed to solve the starvation-free mutual exclusion problem -- global variables with "atomic" read and write operations are powerful enough by themselves. Because of this, it was found to be impossible to learn anything about the relative power of semaphore primitives, unless further restrictions were imposed on the class of mutual exclusion systems. The properties of symmetry and busy-waiting were identified and given precise definitions. It was proved that the classes of symmetric mutual exclusion systems and mutual exclusion systems with no busy-waiting contain no semaphore-free solutions to the starvation-free mutual exclusion problem.

A number of "negative" results about weak semaphore solutions to the starvation-free mutual exclusion problem were proved in Chapter 3. These results, coupled with the "positive" results of Chapter 4, show that weak semaphores are indeed "weaker" in a certain sense than blocked-set semaphores. In addition, it was shown that under certain conditions, weak binary semaphores are strictly less powerful than weak general semaphores.

In Chapter 5 a blocked-set semaphore solution to the starvation-free mutual exclusion problem was proved correct. There appeared to be no short, elegant way to perform this proof. On the contrary, rather long formulas, and tedious verification of many cases were required. What made the proof manageable was the use of the induction principle and the notion of a parallel program homomorphism to make a clean separation of the mechanical parts of the proof from the intuitive parts.

6.2 Directions for Further Study

There are a number of ways in which the work presented in this thesis might be improved upon or extended. We will first examine some possible improvements, and then take a look at extensions.

6.2.1 Possible Improvements

One place where improvement might be made is in notation. At times during the writing of this thesis, there seemed to be just too many things to name. This problem, which seems to be common to other attempts at formal modeling of concurrent processes, may be due to the fact that all discussion takes place at a very low level, at which each minute step of every process must be explicitly represented (and named). No convenient way has been discovered to raise the level of discussion to a higher level, with consequent reduction in the number of details. It is difficult to tell how much of the complexity is inherent in the highly interactive nature of the computations being discussed, and how much could be avoided through better techniques.

There is no good reason why the finite delay property and the fact that a process may have noncritical regions that do not terminate should be intertwined in the definition of valid execution sequences. The fact that in this thesis the critical and noncritical regions have been "collapsed" down to a single place leads to this unintuitive combination of two unrelated notions. Intuitively, the finite delay property applies to processes with nontrivial critical and noncritical regions. Perhaps a formulation of the mutual exclusion problem that somehow retained the idea that critical and noncritical regions in general have more than one place would admit a better definition of valid execution sequences.

The definition of systems of processes and mutual exclusion systems by making syntactic restrictions on parallel program graphs is not entirely satisfactory. Since graphs are not very "nice" to work with, these restrictions did not turn out to be especially elegant. Perhaps there is a cleaner formulation of the model that avoids some of these problems. It is interesting to note that the results proved in Chapter 3 seem to express truths that are somewhat independent of the details of the model used.

6.2.2 Possible Extensions

There are a number of questions left unanswered by the work presented here. The most obvious ones involve straightforward extensions of the results proved in Chapter 3.

(1) It is unknown whether there is a symmetric solution, using weak general semaphores and with no memory, to the starvation-free mutual exclusion problem for more than two processes.

(2) It would also be interesting to examine semaphore-free solutions in more detail. Unanswered questions here are: How many global variables are required to implement starvation-free mutual exclusion for N processes? It is not difficult to show that two processes require at least two variables, and it seems intuitive that the number of variables should increase monotonically with the number of processes, but how can this be proved? Semaphore-free solutions must be asymmetric. Consequently, such solutions are always presented in "parameterized" form; that is, the program run by process i depends upon the number i. The question arises: must these solutions also be parameterized by the total number of processes N as well? It seems as though it might be possible to use the "lock-step" construction to prove that this must be the case. For the "lock-step" construction to be applicable to asymmetric solutions requires increasing the number of processes in the system large

enough to "match-up" initial segments of execution sequences for various processes.

(3) Intuitively, weak semaphores, should be the "weakest" at implementing starvation-free mutual exclusion, blocked-set semaphores somewhat "stronger" and blocked-queue semaphores the "strongest". This intuition is borne out by the fact that in each of the solutions presented in this thesis, weak semaphore operations may be replaced by blocked-set operations, and blocked-set operations may be replaced by blocked-queue operations, and the solution remains correct. However, it is not at all clear that this is true in general. The reason is that replacing semaphore operations in the manner described, "restricts" the set of execution sequences of the system in a certain sense. While it can be shown that the new mutual exclusion system has the mutual exclusion property, is free from indefinite postponement, and is starvation-free if these properties held for the original system, it is not so obvious how to show that "restricting" the set of execution sequences of a mutual exclusion system does not introduce the possibility of deadlock. Thus, based on the results of this thesis, we are unable to justify our intuition that weak, blocked-set, and blocked-queue semaphores may be linearly ordered by their expressive power.

(4) There are other unanswered questions that are not quite so clearly defined. For example, when are two definitions of semaphore operations "equivalent"? This question is important in determining the generality of the results of this thesis. An interesting possibility for future research, which might answer this question, is the development of what might be called a theory of specification and implementation of parallel programs. In such a theory, synchronization behavior would be specified by a parallel program, along with some additional restrictions on the set of execution sequences of that program. These additional restrictions may be necessary since there are interesting constraints, such as the finite delay property, on the execution sequences of a parallel program that cannot be enforced within the parallel program.

A second parallel program would be said to *implement* the abstract behavior specified by the first if there were a suitable parallel program homomorphism from the second program to the first. This idea is quite similar to the notions of "abstract" synchronization primitives and their implementations discussed in [DOEPP76].

One of the goals of such a theory would be the ability to treat synchronization constructs as primitive at one level of abstraction, and to consider their possible implementations at another level. For example, at one level it might be useful to consider several possible implementations of weak semaphore operations, and to prove that the implementations actually satisfy the specifications of the behavior of weak semaphores. At a higher level, it should be possible to treat weak semaphores as primitives, and use them to implement starvation-free mutual exclusion. At yet a higher level, the starvation-free critical regions would themselves be used as primitives, perhaps for implementing a more complicated synchronization scheme, such as hierarchical locks in a database system. It should be possible to treat each of these different levels of abstraction with the same methods.

6.3 Conclusion

The major contribution of this thesis is that it brings the murky issues of "fairness" often mentioned in the synchronization literature into sharper focus. The attempt at precise definitions of the various types of semaphores, while perhaps imperfect, helps to clear up confusion that has resulted from informal discussion. In the literature, many pages have been, and continue to be spent in arguments over whether one program solves or does not solve a particular synchronization problem. Often such arguments are useless, since precise specifications are lacking, for the synchronization problem itself, and for what it means to "solve" that problem. It is hoped that this thesis makes a small step toward the resolution of this difficulty.

7. Appendix

The purpose of this Appendix is to give details of the proofs of Claims 5.4 and 5.9, which state that the predicate [*inv*] is invariant for **Σ**, and that the mapping *h* is a parallel program homomorphism from **Σ** to **Δ**.

7.1 Philosophy of the Presentation

There is an unfortunate difficulty that arises with proofs about programs, and especially with proofs about parallel programs. In ordinary mathematics, the proof of a theorem simultaneously provides both an argument for the correctness of the theorem, and intuition into why the theorem is correct. In proving properties of programs, often all or most of the intuition about why the program is correct goes into constructing the set of statements to be proved. Once this set of statements has been constructed, the proof of the program simply consists of the listing of the set of statements, and an often lengthy, mechanical verification of their truth. The result is that the understanding of the program has been divorced from the actual correctness argument.

This is the case with the proofs to be presented here. The constructions of the invariant [*inv*], the parallel program **Δ**, and the mapping *h* require an understanding of the operation of the system **Σ**. Once *inv*, **Δ**, and *h* have been constructed, what remains to be performed is simply a tedious, mechanical verification. It is necessary to perform this verification to make sure that *inv*, **Δ**, and *h* have been defined correctly, however this verification is something that may be performed by a machine, and is not at all interesting to read.

The inductive step in the proof of the invariance of [*inv*] is divided into thirteen major subproofs, corresponding to each of the thirteen disjuncts of *inv*. Each major subproof is divided into forty-one minor subproofs, corresponding to the forty-one transitions in the graph for the *i*th process in Σ. This results in a total of five hundred thirty-three statements to be verified. Most of these statements are trivial to prove, however the difficulty, at least for a human verifier, is not in the actual production of the proofs of the statements, but in differentiating the statements that are trivial to prove from the ones that really require substantial inferences.

In this Appendix, the following philosophy is therefore adopted to guide the presentation: Due to the large number of cases, the only way to raise the reader's level of confidence in the proof to an acceptable level is for him to actually verify each of the statements, either by hand or by machine. An acceptable level of confidence will *not* be achieved simply by reading a proof. The arguments here are therefore intended not to comprise a complete "readable" proof, but to indicate where major inferences are required; making it a reasonably short step to a complete formal verification with a suitable theorem-proving program.

7.2 Proof Outline

Recall that a proof by the induction principle of the invariance of the predicate [*inv*] requires: (Base) that [*inv*] be shown to hold in the initial state; and (Induction Step) a proof that *inv* holds in state $nxt(q, t)$, under the assumption that t is enabled in state q and that *inv* holds in state q. The mapping h can be shown to be a parallel program homomorphism from Σ to Δ by: (Base) showing that $h(q_I) =$ r_I; and (Induction Step) showing, under the assumption that t is enabled in state q, either that $h(q) = h(nxt(q, t))$ or that there exists a unique u such that $h(nxt(q, t)) = nxt(h(q), u)$. The similar structure of the proof of the invariance of $[inv]$ and the proof that h is a parallel program homomorphism from Σ to Δ makes it possible for them to be performed simultaneously.

The base of both proofs is trivial; that is, it is easy to see that $[inv](q_I)$, and that $h(q_I) = r_I$ We will therefore concentrate our efforts on the induction step only.

Let E be an expression that defines a predicate $[E]$ on states. We will say that a transition t *affects* E if t affects any of the places or variables appearing in E. Recall that $inv \equiv \vee_{i=I}^{13} (conf_i \wedge vbls_i) \wedge aux$. There are comparatively few transitions that affect aux, and it will therefore be convenient to treat separately the part of the proof involving aux. This is done in Section 7.3, where it is proved that if $[aux](q)$ is true, and t is enabled in state q, then $[aux](nxt(q, t))$ is true. Many more transitions affect $\vee_{i=I}^{13} (conf_i \wedge vbls_i)$, and therefore for the proof that this predicate is true in state $nxt(q, t)$ to be feasible, some further organization is required.

As discussed in Chapter 5, a state q satisfying $[inv]$ must satisfy exactly one of $conf_I, \ldots, conf_{13}$. Firing a transition in Σ may be viewed as a progression from a configuration of tokens in one of the thirteen categories to a configuration in another. If we know, for example, that state q satisfies $[conf_I]$, then in proving that $[inv](q)$ implies $[inv](nxt(q, t))$, we may use information about the configuration of tokens in state q to reduce the number of transitions t that must be considered. For example, since no processes are in the leaving region in any state q that satisfies $[conf_I]$, we need not be concerned about any transitions t in the leaving region, since these transitions cannot be enabled if $[conf_I]$ is true, and hence the implication to be proved is trivial.

We divide the proof into thirteen major subproofs, corresponding to each of the thirteen different configurations of tokens. In the *i*th subproof, we assume that q is a state satisfying $[conf_i \wedge vbls_i \wedge aux]$, and attempt to show that if t is an arbitrary transition enabled in state q, then $nxt(q, t)$ satisfies $[\vee_{i=1}^{13} (conf_i \wedge vbls_i)]$. Note that to do this, we need only show that $nxt(q, t)$ satisfies $[conf_i \wedge vbls_i]$ for some $1 \leq i \leq 13$. It will not be important to distinguish between transitions in different processes, and since there are forty-one transitions in the graph for each process, we have the forty-one minor subproofs mentioned above.

Note that if in addition we show that either $h(q) = h(nxt(q, t))$, or there is a unique transition u of Δ such that $h(nxt(q, t)) = nxt(h(q), u)$, then we will have also shown that h is a parallel program homomorphism from Σ to Δ.

7.3 Inductive Step for the Auxiliary Invariants

In this section, we will show that if q is a state satisfying $[inv]$, t is a transition enabled in state q, and $q' = nxt(q, t)$, then $[aux](q')$ is true. This will be done in the following way: Recall that $aux \equiv aux_1 \wedge ... \wedge aux_{21}$. Let us say that t *falsifies* aux_i if $[aux_i](q')$ is false. Note that a transition cannot falsify aux_i unless it affects it. Treating each aux_i as a separate case, we will list the transitions that affect aux_i, and show for each one that aux_i is not falsified.

(1) *Case* aux_1*:* The only transitions that affect the variables a, a_en, or a_bl, and hence the only transitions that affect aux_1 are TT8, TT9, TT10, TT21, LT7, and LT8.

- (a) Transition TT8 can only cause a to become zero, and therefore cannot falsify aux_1.
- (b) Transition TT9 is enabled only if $q(a) = 0$, and hence even though firing it causes a_bl to be nonempty, $q'(a) = 0$, and hence aux_1 is true in state q'.
- (c) Transition TT10 is only enabled if a_en is nonempty in state q. By $[aux_1](q)$ this means that $q(a) = 0$ and hence that $q'(a) = 0$, since TT10 does not

affect a.

(d) Since transitions TT22 and LT8 do not change the value of a or the total number of elements in $a\_bl \cup a\_en$, they do not falsify aux_1.

(e) Transition TT21 is enabled only if $conf_4$, $conf_5$, or $conf_6$ is true in state q. This in turn means that either $vbls_4$, $vbls_5$, or $vbls_6$ is true in state q, and hence that $[\neg en(a)](q)$ is true. Since TT21 is enabled only if $q(a\_bl) = \varnothing$, even though TT21 increments a, it does not falsify aux_1, since $q'(a\_bl) = q'(a\_en) = \varnothing$.

(f) Transition LT7 is enabled only if $conf_{13}$ is true in state q. This in turn means that $vbls_{13}$ is true in state q, and hence that $en(a)$ is false in state q. Therefore $a\_en = \varnothing$ in state q, and since m_bl must be empty for LT7 to be enabled, transition LT7 does not falsify aux_1.

(2) *Case* aux_2*:* The only transitions that affect aux_2 are TT1, TT2, TT3, TT6, TT7, TT11, TT12, TT13, TT19, TT20, TT24, and TT25.

(a) Firing either TT1 or TT11 can only cause b to become zero, and hence cannot falsify aux_2.

(b) Transition TT2 or TT12 is enabled only if $q(b) = 0$, and hence even though firing one of them causes b_bl to become nonempty, $q'(b) = 0$, and hence aux_2 is true in state q'.

(c) Transition TT3 or TT13 is enabled only if b_en is nonempty in state q. By $[aux_2](q)$ this means that $q(b) = 0$ and hence $q'(b) = 0$, since neither TT3 or TT13 affects b.

(d) Since transitions TT7, TT20, and TT25 do not change the value of b or the total number of elements in $b\_bl \cup b\_en$, they do not falsify aux_2.

(e) Transition TT6 is enabled only if $|MUTEX1| = 1$ in state q. Examination of inv reveals that this only occurs when $en(b)$ is false in state q. Also TT19 and TT24 are enabled only if $|MUTEX1| = 1$ in state q, which also implies that *en*(b) is false in state q. Since for TT6, TT19, or TT24 to be enabled in state q requires that $b\_bl = \emptyset$, neither of these transitions can falsify aux_2.

(3) *Case* aux_3*:* This can only be affected by transitions TT28, TT29, TT30, LT4, and LT5.

(a) Firing transition TT28 can only cause m to become zero, and therefore cannot falsify aux_3.

(b) Transition TT29 is enabled in state q only if $q(m) = 0$, and hence even though firing it causes m_bl to become nonempty, $q'(m) = 0$, and hence aux_3 is true in state q'.

(c) Transition LT5 is enabled in state q only if m_en is nonempty. By $[aux_3](q)$ this means that $q(m) = 0$ and hence $q'(m) = 0$, since LT5 does not affect m.

(d) Since TT30 does not change the value of m, or the total number of elements in $m\_bl \cup m\_en$, it does not falsify aux_3.

(e) Transition LT4 is enabled in state q only if $conf_{13}$ is true in state q. This means that $vbls_{13}$ must be true in state q, and hence *en*(m) is false. Thus m_en is empty in state q. Since for LT4 to be enabled requires m_bl to be empty in state q, we see that *LT4* cannot falsify aux_3.

(4) *Case* aux_4*:* Transitions TT8, TT21, and LT7 affect aux_4. Transition TT8 decrements a, and therefore cannot falsify aux_4. It is easily verified by examination of *inv* that TT21 and LT7 are enabled only if *en*(a) is false in state q. But this implies that $q(a) = 0$. Hence $q'(a) = 1$, and aux_4 is not falsified by TT21 or LT7.

(5) *Case* aux_5*:* Transitions TT1, TT6, TT11, TT19, and TT24 affect aux_5. The arguments are similar to those of case (4).

(6) *Case* aux_6*:* Transitions TT28 and LT4 affect aux_6. The arguments are similar to those of case (4).

(7) *Case aux_7:* Transitions TT10, TT22, and LT8 affect the variable *a_en*, and hence affect aux_7. Transition TT10 decreases the number of elements in *a_en*, and therefore cannot falsify aux_7. Examination of *inv* shows that TT22 and LT8 are enabled in state *q* only if *en(a)* is false in state *q*, and hence only if $q(a\_en) = \emptyset$. Since these transitions add exactly one element to *a_en*, they therefore cannot falsify aux_7.

(8) *Case aux_8:* Transitions TT3, TT7, TT13, TT20, and TT25 affect the variable *b_en*, and hence affect aux_8. An argument similar to that in case (7) shows that these transitions do not falsify aux_8.

(9) *Case aux_9:* Transitions TT30 and LT5 affect the variable *m_en*, and hence affect aux_9. An argument similar to that in case (7) shows that these transitions do not falsify aux_9.

(10) *Case aux_{10}:* Transitions TT9, TT10, TT22, and LT8 affect aux_{10}. Transitions TT22 and LT8 do not change the elements in $a\_en \cup a\_bl$, and hence do not falsify aux_{10}. Transition TT9 inserts a process number into *a_bl*, but at the same time the token for that process enters place TP7. Similarly, transition TT10 removes a process number from *a_en*, but at the same time removes the corresponding token from TP7. Thus transitions TT9 and TT10 do not falsify aux_{10}.

(11) *Case aux_{11}:* Transitions TT2, TT3, TT7, TT12, TT13, TT20, and TT25 affect aux_{11}. An argument similar to that in case (10) shows that these transitions do not falsify aux_{11}.

(12) *Case aux_{12}:* Transitions TT29, TT30, and LT8 affect aux_{12}. An argument similar to that in case (10) shows that these transitions do not falsify aux_{12}.

(13) *Case* aux_{13}: Transitions TT9, TT10, TT22, and LT8 affect aux_{13}. Now, transitions TT22 and LT8 simply transfer a process number from *a_bl* to *a_en*, and therefore cannot falsify aux_{13}. Transition TT10 removes an element from *a_en*, and therefore cannot falsify aux_{13} either. Finally, TT9 cannot falsify aux_{13} either, since by $[aux_{10}](q)$ a process whose token is at place TP6 in state q cannot have its process number in *a_en* in state q.

(14) *Case* aux_{14}: Transitions TT2, TT3, TT7, TT12, TT13, TT20, and TT25 affect aux_{14}. An argument similar to that of case (13) shows that none of these transitions can falsify aux_{14}.

(15) *Case* aux_{15}: Transitions TT29, TT30, and LT5 affect aux_{15}. An argument similar to that of case (13) shows that none of these transitions can falsify aux_{15}.

(16) *Case* aux_{16}: Transitions TT8, TT10, and TT13 affect aux_{16}. However, transition TT13 decreases the number of processes at *WAITB2*, and hence does not falsify aux_{16}. Also, for transition TT8 or TT10 to be enabled, we must have that $en(a)$ is false in state q. Examination of *inv* reveals that this can only happen when $|WAITB2| = 0$ in state q, and hence TT8 and TT10 do not falsify aux_{16} either.

(17) *Case* aux_{17}: Transitions TT1, TT3, TT6, and TT7 affect aux_{17}. However, transitions TT6 and TT7 decrease the number of processes at *MUTEX1*, and hence cannot falsify aux_{17}. Also, for transition TT1 or TT3 to be enabled in state q, we must have that $en(b)$ is false in state q. Examination of *inv* reveals that this can only happen if $|MUTEX1| = 0$ in state q, and hence TT1 and TT3 do not falsify aux_{17} either.

(18) *Case* aux_{18}: Transitions TT5, ... , TT15 affect aux_{18}. However, aux_{18} is obviously not falsified by transitions TT6, ... , TT14, since these transitions change

neither $|TP5| + ... + |TP11|$ nor the variable *count1*. Although transition TT5 increases $|TP5| + ... + |TP11|$ by one, for TT5 to be enabled in state q means that there is at least one process at *TP4* in state q, and hence by $[aux_{19}](q)$ we know that $loc\_i = count1$ in state q. Therefore firing TT5 also increases *count1* by one, and does not falsify aux_{18}. Similarly, although TT15 decreases $|TP5| + ... + |TP11|$, firing that transition also decreases *count1*, and therefore does not falsify aux_{18}.

(19) *Case* aux_{19}: Transitions TT4 and TT14 affect aux_{19}, but obviously do not falsify it.

(20) *Case* aux_{20}: The argument here is similar to that of case (18).

(21) *Case* aux_{21}: The argument here is similar to that of case (19).

7.4 Remainder of the Inductive Step

In the previous section, we showed that if q is a state satisfying $[inv]$, t is a transition enabled in state q, and $q' = nxt(q, t)$, then q' satisfies $[aux]$. To complete the proof of Claims 5.4 and 5.9, we must show, under the same assumptions, that q' satisfies $[\vee_{i=1}^{13} (conf_i \wedge vbls_i)]$, and that either $h(q) = h(q')$, or there is a unique transition u of Δ such that $h(q') = nxt(h(q), u)$. As previously mentioned, the proof will be split into thirteen major subproofs, where in the ith subproof the additional assumption is made that $[conf_i \wedge vbls_i](q)$ is true. Each of these subproofs will be divided into forty-one minor subproofs; one for each of the forty-one transitions in the graph for process i.

The statements to be proved are summarized in tabular form in Figure 7.1. There are four columns in Figure 7.1. The "INDEX" column assigns an identifying index to each case for reference purposes. These indices are of the form N1-T(-N2), where N1 is a number from one to thirteen indicating the major subproof to which the case belongs, and the minor subproof is indicated by T, which is a transition name. The parentheses indicate that the third field N2 is optional. This field is present only when a further division into subcases must be made; more will be said about this below.

To see how each row of the table indicates a statement to be proved, let us use the row with index 2-TT6-1 as an example. The index indicates that we are to assume that $[conf_2 \wedge vbls_2 \wedge aux](q)$ is true, and consider the firing of transition TT6. The "Additional Assumptions" column contains a set of additional assumptions about the state q which we also make. In this case, the additional assumptions are that $[|WAITA| + |WAITB1| + |WAITB2| > 0](q)$ is true. Note that the rows with indices 2-TT6-1 and 2-TT6-2 represent a partitioning of minor subproof 2-TT6 into two disjoint cases, defined by the "Additional Assumptions" information.

The "New Conf" column contains the entry "1", which indicates that we are to show that, under these assumptions, if transition TT6 is enabled in state q and $q' = nxt(q, \text{TT6})$, then $[conf_1 \wedge vbls_1](q')$ is true. If the "New Conf" column for row 2-TT6-1 contained a hypen "-" instead of a "1", then we would be required to show instead that transition TT6 could not be enabled in state q under the stated assumptions.

Finally, the transition of Δ named in the "Δ Trans" column of Figure 7.1 indicates the unique transition u of Δ such that $h(q') = nxt(h(q), u)$. In row 2-TT6-1, this transition is "1a-5". The complete transition label may be found by looking up "1a-5" in Figure 5.7. A hyphen in the "Δ Trans" column indicates that we are to show $h(q) = h(q')$, and hence there is no corresponding transition u of Δ.

Note that many minor subproofs, for example 1-TT4, are not listed in Figure 7.1. This is because it is immediately obvious that transition TT4 cannot be enabled in any state in which $conf_1$ is true, and therefore need not be considered further. A

Index	New Conf	Δ Trans	Additional Assumptions
1-NCT	1	8a-2	
1-TT1	2	1a-2	
1-TT2	1	8a-2	
1-TT3	2	1a-3	
1-TT8	1	1a-1	
1-TT9	1	-	
1-TT10	1	1a-1	
1-TT11	3	1a-4	
1-TT12	1	-	
1-TT13	3	1a-4	
1-TT28	-		
1-TT29	1	-	
1-TT30	-		
1-LT9	1	-	
2-NCT	2	8a-1	
2-TT1	-		
2-TT2	2	8a-2	
2-TT3	-		
2-TT4	2	-	
2-TT5	2	-	
2-TT6-1	1	1a-5	$\lvert WAITA\rvert + \lvert WAITB1\rvert + \lvert WAITB2\rvert > 0$
2-TT6-2	7	1c-1	$\lvert WAITA\rvert + \lvert WAITB1\rvert + \lvert WAITB2\rvert = 0$
2-TT7-1	1	1a-5	$\lvert WAITA\rvert + \lvert WAITB1\rvert + \lvert WAITB2\rvert > 0$
2-TT7-2	7	1c-1	$\lvert WAITA\rvert + \lvert WAITB1\rvert + \lvert WAITB2\rvert = 0$
2-TT8	2	1a-1	
2-TT9	2	-	
2-TT10	2	1a-1	
2-TT11	-		
2-TT12	2	-	
2-TT13	-		
2-TT28	-		
2-TT29	2	-	
2-TT30	-		
2-LT9	2	-	

Figure 7.1 - Exhaustive List of Cases

Index	New Conf	Δ Trans	Additional Assumptions
3-NCT	3	8a-1	
3-TT1	-		
3-TT2	3	8a-2	
3-TT3	-		
3-TT8	-		
3-TT9	3	-	
3-TT10	-		
3-TT14	3	-	
3-TT15	3	-	
3-TT16	3	-	
3-TT17	3	-	
3-TT18	3	-	
3-TT19-1	4	1a-6	$\|WAITA\| \neq 1 \lor \|WAITBI\| \neq 0$
3-TT19-2	5	1b-1	$\|WAITA\| = 1 \land \|WAITBI\| = 0$
3-TT20-1	4	1a-6	$\|WAITA\| \neq 1 \lor \|WAITBI\| \neq 0$
3-TT20-2	5	1b-1	$\|WAITA\| = 1 \land \|WAITBI\| = 0$
3-TT23	-		
3-TT28	-		
3-TT29	3	-	
3-TT30	-		
3-LT9	3	-	
4-NCT	4	8a-1	
4-TT1	6	1a-2	
4-TT2	4	8a-2	
4-TT3	6	1a-3	
4-TT8	-		
4-TT9	4	-	
4-TT10	-		
4-TT21	1	1a-7	
4-TT22	1	1a-7	
4-TT28	-		
4-TT29	4	-	
4-TT30	-		
4-LT9	4	-	

Figure 7.1 - Exhaustive List of Cases (cont.)

Index	New Conf	Δ Trans	Additional Assumptions
5-NCT	5	8a-1	
5-TT1	6	2a-1	
5-TT2	5	8a-2	
5-TT3	6	2a-2	
5-TT8	-		
5-TT9	5	-	
5-TT10	-		
5-TT21	7	2b-1	
5-TT22	7	2b-1	
5-TT28	-		
5-TT29	5	-	
5-TT30	-		
5-LT9	5	-	
6-NCT	6	8a-1	
6-TT1	-		
6-TT2	6	8a-2	
6-TT3	-		
6-TT4	6	-	
6-TT5	6	-	
6-TT6	4	1a-5	
6-TT7	4	1a-5	
6-TT8	-		
6-TT9	6	-	
6-TT10	-		
6-TT21	2	1a-7	
6-TT22	2	1a-7	
6-TT28	-		
6-TT29	6	-	
6-TT30	-		
6-LT9	6	-	
7-NCT	7	8a-1	
7-TT1	2	3b-1	
7-TT2	7	8a-2	
7-TT3	2	3b-2	

Figure 7.1 - Exhaustive List of Cases (cont.)

Index	New Conf	Δ Trans	Additional Assumptions
7-TT8	8	3a-1	
7-TT9	7	-	
7-TT10	8	3a-1	
7-TT28	-		
7-TT29	7	-	
7-TT30	-		
7-LT9	7	-	
8-NCT	8	8a-1	
8-TT1	2	4c-1	
8-TT2	8	8a-2	
8-TT3	2	4c-2	
8-TT11	9	4a-1	
8-TT12	8	-	
8-TT13	9	4a-1	
8-TT28	-		
8-TT29	8	-	
8-TT30	-		
8-LT9	8	-	
9-NCT	9	8a-1	
9-TT1	-		
9-TT2	9	8a-2	
9-TT3	-		
9-TT14	9	-	
9-TT15	9	-	
9-TT16	9	-	
9-TT17	9	-	
9-TT18	-		
9-TT23	9	-	
9-TT24	10	4a-5	
9-TT25	10	4a-5	
9-TT28	-		
9-TT29	9	-	
9-TT30	-		
9-LT9	9	-	

Figure 7.1 - Exhaustive List of Cases (cont.)

Index	New Conf	Δ Trans	Additional Assumptions
10-NCT	10	8a-1	
10-TT1	10	4a-2	
10-TT2	10	8a-2	
10-TT3	10	4a-3	
10-TT4	10	-	
10-TT5	10	-	
10-TT6	10	4a-4	
10-TT7	10	4a-4	
10-TT8	-		
10-TT9	10	-	
10-TT10	-		
10-TT26	11	4b-1	
10-TT27	11	4b-1	
10-TT28	-		
10-TT29	10	-	
10-TT30	-	-	
10-LT9	10	-	
11-NCT	12	8a-1	
11-TT1	11	5a-1	
11-TT2	11	8a-2	
11-TT3	11	5a-2	
11-TT4	11	-	
11-TT5	11	-	
11-TT6	11	5a-3	
11-TT7	11	5a-3	
11-TT8	-		
11-TT9	11	-	
11-TT10	-		
11-TT28	12	5b-1	
11-TT29	11	-	
11-TT30	12	5b-1	
11-LT9	11	-	
12-NCT	12	8a-1	
12-TT1	12	6a-1	

Figure 7.1 - Exhaustive List of Cases (cont.)

Index	New Conf	Δ Trans	Additional Assumptions
12-TT2	12	8a-2	
12-TT3	12	6a-2	
12-TT4	12	-	
12-TT5	12	-	
12-TT6	12	6a-3	
12-TT7	12	6a-3	
12-TT8	-		
12-TT9	12	-	
12-TT10	-		
12-TT28	-		
12-TT29	12	-	
12-TT30	-		
12-CRT	13	6b-1	
12-LT9	12	-	
13-NCT	13	8a-1	
13-TT1	13	7a-1	
13-TT2	13	8a-2	
13-TT3	13	7a-2	
13-TT4	13	-	
13-TT5	13	-	
13-TT6	13	7a-3	
13-TT7	13	7a-3	
13-TT8	-		
13-TT9	13	-	
13-TT10	-		
13-TT28	-		
13-TT29	13	-	
13-TT30	-		
13-LT1	13	-	
13-LT2	13	-	
13-LT3	13	-	
13-LT4	11	7b-1	
13-LT5	11	7b-1	
13-LT6	13	-	

Figure 7.1 - Exhaustive List of Cases (cont.)

Index	New Conf	Δ Trans	Additional Assumptions
13-LT7-1	1	7c-1	$\lvert MUTEXI\rvert = 0 \land (\lvert WAITA\rvert = 1 \rightarrow \mathit{procnotcounted})$
13-LT7-2	2	7c-1	$\lvert MUTEXI\rvert > 0$
13-LT7-3	7	7d-1	$\lvert MUTEXI\rvert = 0 \land \lvert WAITA\rvert = 1 \land \neg \mathit{procnotcounted}$
13-LT8-1	1	7c-1	$\lvert MUTEXI\rvert = 0 \land (\lvert WAITA\rvert = 1 \rightarrow \mathit{procnotcounted})$
13-LT8-2	2	7c-1	$\lvert MUTEXI\rvert > 0$
13-LT8-3	3	7d-1	$\lvert MUTEXI\rvert = 0 \land \lvert WAITA\rvert = 1 \land \neg \mathit{procnotcounted}$
13-LT9	13	-	

Figure 7.1 - Exhaustive List of Cases (cont.)

similar observation holds for all other missing cases.

The verification, for each row in Figure 7.1, that $[conf_j](q')$ is true for the number j indicated in the "New Conf" column, is trivial. It is also not difficult to verify for each case that either $h(q) = h(q')$ or $h(q') = nxt(h(q), u)$, where u is the transition indicated in the "Δ Trans" column. Consequently, these two arguments are left to the reader. The only part of the argument for each case that requires substantial inference is showing that $[vbls_j](q')$ is true for the number j indicated in the "New Conf" column. These arguments are presented below.

1-NCT: Transition NCT does not affect $vbls_1$.

1-TT1: Of the terms in $vbls_1$, namely $|WAITA|$, $|WAITB1|$, $|WAITB2|$, $|WAITM|$, $en(a)$, $en(b)$, $en(m)$, *procnotcounted*, the only one that is affected by TT1 is $en(b)$. If we can show that $[\neg en(b)](q')$ is true, then the truth of $vbls_2$ in state q' follows directly from the truth of $vbls_1$ in state q, and the fact that only $en(b)$ is affected by TT1. Now, for TT1 to be enabled in state q, we must have $q(b) > 0$. Because $[aux_2](q)$ and $[aux_5](q)$ are true, we know that $q(b) = 1$, and that $q(b\_en) = \varnothing$. Since $q'(b) = q(b) - 1 = 0$, we have that $[\neg en(b)](q')$ is true.

1-TT2: The only terms in $vbls_1$ affected by TT2 are *procnotcounted* and $|WAITB1|$. Now, TT2 increments $|WAITB1|$, and therefore $|WAITB1| + |WAITA| + |WAITB2|$ is greater than zero in state q'. Also, although firing TT2_i moves the token for process i to *WAITB1*, this cannot falsify *procnotcounted*, since TT2 does not change *b_en*, and by $[aux_{11}](q)$, the number i is not in *b_en* in state q.

1-TT3: Of the terms in $vbls_1$, the only one affected by TT3 is $en(b)$. However, TT3 can be enabled in state q only if $q(b\_en) \neq \varnothing$. But by $[aux_2](q)$ and $[aux_8](q)$ we know that there is exactly one element in *b_en* in state q and that $q(b) = 0$. Since firing TT3 removes an element from b_en, we have that $[\neg en(b)](q')$ is true. The truth of $vbls_2$ in state q' now follows from the truth of $vbls_1$ in state q and the fact that only $en(b)$ is affected by TT3.

1-TT8: Since aux_1 and aux_4 are true in state q by hypothesis, and since for TT8 to be enabled in state q requires $q(a) > 0$, it must be the case that $q(a) = 1$ and $q(a\_en) = \varnothing$. Since TT8 decrements a, $[\neg en(a)](q')$ is true. In addition, by $[aux_{16}](q)$ and $[vbls_1](q)$ we know that $|WAITB2| = 0$ in state q. Since firing TT8 increments $|WAITB2|$ we know that $|WAITB2| = 1$ in state q'. Since TT8 affects only the terms $en(a)$ and $|WAITB2|$, it is clear that $vbls_1$ is true in state q'.

1-TT9: Transition TT9 does not affect $vbls_1$.

1-TT10: The fact that $en(a)$ is false in state q' follows from $[aux_1](q)$, $[aux_7](q)$, and from the fact that for TT10 to be enabled in state q requires $q(a\_en) \neq \varnothing$. The rest of the argument is identical to case 1-TT8.

1-TT11: For TT11 to be enabled in state q, it must be the case that $q(b) > 0$ and $|WAITB2| > 0$ in state q. Hence, by $[vbls_1](q)$ it must be that $[\neg en(a) \wedge en(b) \wedge \neg en(m)](q)$ is true. Since TT11 does not affect $en(a)$ or $en(m)$, we only need to show that $en(b)$ is false in state q'. But this follows from $[aux_2 \wedge aux_5](q)$ as in case 1-TT1.

1-TT12: Transition TT12 does not affect $vbls_1$.

1-TT13: For TT13 to be enabled in state q, it must be the case that $q(b\_en) \neq \varnothing$, and $|WAITB2| > 0$ in state q. Hence, by $[vbls_1](q)$ it must be that $[\neg en(a) \wedge en(b) \wedge \neg en(m)](q)$ is true. Since TT13 does not affect $en(a)$ or $en(m)$, we only need to show that $en(b)$ is false in state q'. But this follows from $[aux_2 \wedge aux_8](q)$ as in case 1-TT3.

1-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

1-TT29: Transition TT29 does not affect $vbls_1$.

1-TT30: Impossible since TT20 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

1-LT9: Transition LT9 does not affect $vbls_1$.

2-NCT: Transition NCT does not affect $vbls_2$.

2-TT1: Impossible since TT1 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

2-TT2: Transition TT2 does not affect $vbls_2$.

2-TT3: Impossible since TT3 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

2-TT4: Transition TT4 does not affect $vbls_2$.

2-TT5: Transition TT5 does not affect $vbls_2$.

2-TT6-1: We must show that $[vbls_1](q')$ is true. Note first that since TT6 increments b, $en(b)$ is true in state q'. Transition TT6 increments $|WAITA|$, and so $|WAITB1| + |WAITA| + |WAITB2|$ is greater than zero in state q'. Transition TT6 does not affect $en(m)$, $en(a)$, or $|WAITB2|$. It therefore remains to be shown that $|WAITB2| = 0 \rightarrow (|WAITA| = 1 \rightarrow procnotcounted)$ and $|WAITB2| = 1 \rightarrow (|WAITA| = 0 \rightarrow procnotcounted)$ are true in state q'.

By the assumption distinguishing case 2-TT6-1 from 2-TT6-2, one of $|WAITA|$, $|WAITB1|$, or $|WAITB2| > 0$ in state q. If $|WAITB2| > 0$ in state q, and hence in state q', then since $[|WAITA|](q') > 0$, $[vbls_1](q')$ is satisfied. Otherwise, if $[|WAITB2|](q) = 0$, then there are two cases: either $[|WAITA|](q) > 0$ or $[|WAITA|](q) = 0$. In the first case, $[vbls_1](q')$ is satisfied since $[|WAITA|](q') > 1$. In the second case, it must be that $|WAITB1| > 0$ in state q. By the truth of aux_{11} in state q, and the fact that for TT6 to be enabled in state q requires $q(b\_bl) = 0$, we know that $[procnotcounted](q')$ is true. Hence $[vbls_1](q')$ is true.

2-TT6-2: We must show that $[vbls_7](q')$ is true. The truth of $en(a)$, $en(b)$, and $\neg en(m)$ in state q' is easily shown as in case 2-TT6-1. Since $|WAITB1| = 0$ in state q by the assumption distinguishing case 2-TT6-2, and TT6 does not affect $|WAITB1|$, we have that $[\neg procnotcounted](q')$ is true, and hence $[vbls_7](q')$ is true.

2-TT7: Firing TT7 adds an element to b_en, so we know that $[en(b)](q')$ is true. Since TT7 can be enabled only if $q(b\_bl) \neq \varnothing$, we know from $[aux_{11}](q)$ that $|WAITB1| + |WAITB2| > 0$ in state q. Proving that $vbls_1$ is true in state q' may now be done as in case 2-TT6-1.

2-TT8: This case is similar to 1-TT8.

2-TT9: Transition TT9 does not affect $vbls_2$.

2-TT10: This case is similar to 1-TT10.

2-TT11: Impossible since TT11 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

2-TT12: Transition TT12 does not affect $vbls_2$.

2-TT13: Impossible since TT13 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

2-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

2-TT29: Transition TT29 does not affect $vbls_2$.

2-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

2-LT9: Transition LT9 does not affect $vbls_2$.

3-NCT: Transition NCT does not affect $vbls_3$.

3-TT1: Impossible since TT1 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

3-TT2: Transition TT2 does not affect $vbls_3$.

3-TT3: Impossible since TT3 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

3-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

3-TT9: Transition TT9 does not affect $vbls_3$.

3-TT10: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

3-TT14: Transition TT14 does not affect $vbls_3$.

3-TT15: Transition TT15 does not affect $vbls_3$.

3-TT16: Transition TT16 does not affect $vbls_3$.

3-TT17: Transition TT17 does not affect $vbls_3$.

3-TT18: Transition TT18 does not affect $vbls_3$.

3-TT19-1: The truth of $[vbls_3](q)$ implies that $en(a)$ and $en(m)$ are false in state q, and these terms are not affected by TT19. The truth of $[\neg en(b)](q')$ may be established by an argument similar to that of case 1-TT1. If $|WAITA| \neq 1$ in state q, then $[vbls_4](q')$ follows easily. If $|WAITB1| \neq 0$ in state q, then by the truth of aux_{11} in state q, and the fact that for TT19 to be enabled in state q requires $q(b\_bl) = \varnothing$, we know that *procnotcounted* is true in state q, and hence in state q'. Therefore $[vbls_4](q')$ is true.

3-TT19-2: We may establish $[\neg en(a) \wedge en(b) \wedge \neg en(m)](q')$ as in case 3-TT19-1. Since $|WAITB1| = 0$ in state q, and hence in state q', we know that *procnotcounted* is false in state q', and therefore that $[vbls_5](q')$ is true.

3-TT20: For TT20 to be enabled in state q, it must be the case that $q(b\_bl) \neq \varnothing$. By the truth of aux_{11} in state q, and because $[conf_3](q)$ implies that $|WAITB2| = 0$ in state q, we have that $\forall i((i \text{ at } TP2) \leftrightarrow i \in b\_bl)$ is true in state q. The effect of firing TT20 is to remove an element from b_bl and transfer it to b_en. But this means that *procnotcounted* is true in state q'. The remainder of the proof of $[vbls_4](q')$ is done as in case 3-TT19-1.

3-TT23: Impossible, because $conf_3$ true in state q means that $|MUTEX2| = 1$ in state q. By the truth of aux_{18} in state q, we know that therefore $q(count1) > 0$, and hence TT23 cannot be enabled in state q.

3-TT28: Impossible because TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

3-TT29: Transition TT29 does not affect $vbls_3$.

3-TT30: Impossible because TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

3-LT9: Transition LT9 does not affect $vbls_3$.

4-NCT: Transition NCT does not affect $vbls_4$.

4-TT1: Verifying $[\neg en(b)](q')$ may be done as in case 1-TT1.

4-TT2: Transition TT2 does not affect $vbls_4$.

4-TT3: Verifying $[\neg en(b)](q')$ may be done as in case 1-TT3.

4-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

4-TT9: Transition TT9 does not affect $vbls_4$.

4-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

4-TT21: *Procnotcounted* is not affected by TT21. Since TT21 increments a, we have that $en(a)$ is true in state q'. Finally, $|WAITA| > 0$ in state q by $conf_4$, and is not affected by TT21.

4-TT22: Similar to case 4-TT21, except that $[en(a)](q')$ is verified by noting that TT22 adds an element to a_en.

4-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

4-TT29: Transition TT29 does not affect $vbls_4$.

4-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

4-LT9: Transition LT9 does not affect $vbls_4$.

5-NCT: Transition NCT does not affect $vbls_5$.

5-TT1: We may verify $[\neg en(b)](q')$ as in case 1-TT1.

5-TT2: Transition TT2 does not affect $vbls_5$.

5-TT3: We may verify $[\neg en(b)](q')$ as in case 1-TT3.

5-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

5-TT9: Transition TT9 does not affect $vbls_5$.

5-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

5-TT21: Since TT21 increments a, $[en(a)](q')$ is true.

5-TT22: Since TT21 adds an element to a_en, $[en(a)](q')$ is true.

5-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

5-TT29: Transition TT29 does not affect $vbls_5$.

5-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

5-LT9: Transition LT9 does not affect $vbls_5$.

6-NCT: Transition NCT does not affect $vbls_6$.

6-TT1: Impossible since TT1 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

6-TT2: Transition TT2 does not affect $vbls_6$.

6-TT3: Impossible since TT3 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

6-TT4: Transition TT4 does not affect $vbls_6$.

6-TT5: Transition TT5 does not affect $vbls_6$.

6-TT6: Since TT6 increments b, $en(b)$ is true in state q'. Since $conf_6$ is true in state q we know that $|WAITA| > 0$ in state q and hence that $|WAITA| > 1$ in state q'. Therefore $vbls_4$ is true in state q'.

6-TT7: Similar to case 6-TT6 except that $[en(b)](q')$ is true because TT7 adds an element to b_en.

6-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

6-TT9: Transition TT9 does not affect $vbls_6$.

6-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

6-TT21: $|WAITB2| = 0$ in state q, and is not affected by TT21. Since TT21 increments a, $en(a)$ is true in state q.

6-TT22: $|WAITB2| = 0$ in state q, and is not affected by TT22. Since TT22 adds an element to a_en, $en(a)$ is true in state q.

6-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

6-TT29: Transition TT29 does not affect $vbls_6$.

6-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

6-LT9: Transition LT9 does not affect $vbls_6$.

7-NCT: Transition NCT does not affect $vbls_7$.

7-TT1: We may verify $[\neg en(b)](q')$ as in case 1-TT1. Also, by $[conf_7](q)$, $|WAITB2| = 0$ in state q and is not affected by TT2. Since $en(a)$ is true in state q and is not affected by TT2 we have that $vbls_2$ is true in state q'.

7-TT2: Transition TT2 does not affect $vbls_7$.

7-TT3: The argument is identical to that of case 7-TT1, except that $[\neg en(b)](q')$ is verified as in case 1-TT3.

7-TT8: We may verify $[\neg en(a)](q')$ as in case 1-TT8.

7-TT9: Transition TT9 does not affect $vbls_7$.

7-TT10: We may verify $[\neg en(a)](q')$ as in case 1-TT10.

7-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

7-TT29: Transition TT29 does not affect $vbls_7$.

7-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

7-LT9: Transition LT9 does not affect $vbls_7$.

8-NCT: Transition NCT does not affect $vbls_8$.

8-TT1: We may verify $[\neg en(b)](q')$ as in case 1-TT1. Also, $|WAITB2| = 1$ in state q by $[conf_8](q)$, and TT1 does not affect $|WAITB2|$. Hence $[vbls_2](q')$ is true.

8-TT2: Transition TT2 does not affect $vbls_8$.

8-TT3: The argument is identical to that of case 8-TT1, except that $[\neg en(b)](q')$ is verified as in case 1-TT3.

8-TT11: We may verify $[\neg en(b)](q')$ as in case 1-TT11.

8-TT12: Transition TT12 does not affect $vbls_8$.

8-TT13: We may verify $[\neg en(b)](q')$ as in case 1-TT13.

8-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

8-TT29: Transition TT29 does not affect $vbls_8$.

8-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

8-LT9: Transition LT9 does not affect $vbls_8$.

9-NCT: Transition NCT does not affect $vbls_9$.

9-TT1: Impossible since TT1 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

9-TT2: Transition TT2 does not affect $vbls_9$.

9-TT3: Impossible since TT3 cannot be enabled in state q if $[\neg en(b)](q)$ is true.

9-TT14: Transition TT14 does not affect $vbls_9$.

9-TT15: Transition TT15 does not affect $vbls_9$.

9-TT16: Transition TT16 does not affect $vbls_9$.

9-TT17: Transition TT17 does not affect $vbls_9$.

9-TT18: It is impossible for TT18 to be enabled in state q, since $[conf_9](q)$ implies $[|MUTEX1| + |WAITA| + |WAITB2| = 0](q)$, and hence $[aux_{18}](q)$ implies that $q(count1) = 0$.

9-TT23: Transition TT23 does not affect $vbls_9$.

9-TT24: $|MUTEX1| = 0$ in state q, and is not affected by TT24. We may verify $[en(b)](q')$ using $[aux_2](q)$ and $[aux_5](q)$.

9-TT25: The argument is identical to that of case 9-TT24, except that $[en(b)](q')$ is verified using $[aux_2](q)$ and $[aux_8](q)$.

9-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

9-TT29: Transition TT29 does not affect $vbls_9$.

9-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

9-LT9: Transition LT9 does not affect $vbls_9$.

10-NCT: Transition NCT does not affect $vbls_{10}$.

10-TT1: By $[vbls_{10}](q)$ and $[aux_{17}](q)$, TT1 can only be enabled if $|MUTEX1| = 0$ in state q. Firing TT1 then increments $|MUTEX1|$. We may show that $en(b)$ is false in state q' with an argument like that in case 1-TT1.

10-TT2: Transition TT2 does not affect $vbls_{10}$.

10-TT3: The argument is identical to that of case 10-TT1, except that $[\neg en(b)](q')$ is verified as in case 1-TT3.

10-TT4: Transition TT4 does not affect $vbls_{10}$.

10-TT5: Transition TT5 does not affect $vbls_{10}$.

10-TT6: Although TT6 decrements $|MUTEX1|$, it also makes $en(b)$ true in state q', and hence $[vbls_{10}](q')$ is true.

10-TT7: Similar argument to that of case 10-TT6.

10-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

10-TT9: Transition TT9 does not affect $vbls_{10}$.

10-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

10-TT26: Since TT26 increments m, $en(m)$ is true in state q'.

10-TT27: Since TT27 adds an element to m_en, $en(m)$ is true in state q'.

10-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

10-TT29: Transition TT29 does not affect $vbls_{10}$.

10-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

10-LT9: Transition LT9 does not affect $vbls_{10}$.

11-NCT: Transition NCT does not affect $vbls_{11}$.

11-TT1: The argument for this case is similar to that of case 10-TT1.

11-TT2: Transition TT2 does not affect $vbls_{11}$.

11-TT3: The argument for this case is similar to that of case 10-TT3.

11-TT4: Transition TT4 does not affect $vbls_{11}$.

11-TT5: Transition TT5 does not affect $vbls_{11}$.

11-TT6: The argument for this case is similar to that of case 10-TT6.

11-TT7: The argument for this case is similar to that of case 10-TT7.

11-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

11-TT9: Transition TT9 does not affect $vbls_{11}$.

11-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

11-TT28: We may verify $[\neg en(m)](q')$ using $[aux_3](q)$ and $[aux_6](q)$.

11-TT29: Transition TT29 does not affect $vbls_{11}$.

11-TT30: We may verify $[\neg en(m)](q')$ using $[aux_3](q)$ and $[aux_9](q)$.

11-LT9: Transition LT9 does not affect $vbls_{11}$.

12-NCT: Transition NCT does not affect $vbls_{12}$.

12-TT1: The argument for this case is similar to that of case 10-TT1.

12-TT2: Transition TT2 does not affect $vbls_{12}$.

12-TT3: The argument for this case is similar to that of case 10-TT3.

12-TT4: Transition TT4 does not affect $vbls_{12}$.

12-TT5: Transition TT5 does not affect $vbls_{12}$.

12-TT6: The argument for this case is similar to that of case 10-TT6.

12-TT7: The argument for this case is similar to that of case 10-TT7.

12-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

12-TT9: Transition TT9 does not affect $vbls_{12}$.

12-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

12-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

12-TT29: Transition TT29 does not affect $vbls_{12}$.

12-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

12-CRT: Since $|LP4| = 0$ and $|LP5| = 0$ in state q, and this is not affected by CRT, $vbls_{13}$ is true in state q.

12-LT9: Transition LT9 does not affect $vbls_{12}$.

13-NCT: Transition NCT does not affect $vbls_{13}$.

13-TT1: The argument for this case is similar to that of case 10-TT1.

13-TT2: Transition TT2 does not affect $vbls_{13}$.

13-TT3: The argument for this case is similar to that of case 10-TT3.

13-TT4: Transition TT4 does not affect $vbls_{13}$.

13-TT5: Transition TT5 does not affect $vbls_{13}$.

13-TT6: The argument for this case is similar to that of case 10-TT6.

13-TT7: The argument for this case is similar to that of case 10-TT7.

13-TT8: Impossible since TT8 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

13-TT9: Transition TT9 does not affect $vbls_{13}$.

13-TT10: Impossible since TT10 cannot be enabled in state q if $[\neg en(a)](q)$ is true.

13-TT28: Impossible since TT28 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

13-TT29: Transition TT29 does not affect $vbls_{13}$.

13-TT30: Impossible since TT30 cannot be enabled in state q if $[\neg en(m)](q)$ is true.

13-LT1: Transition LT1 does not affect $vbls_{13}$.

13-LT2: Transition LT2 does not affect $vbls_{13}$.

13-LT3: Even though $|LP4|$ becomes positive in state q', by the truth of $[aux_{20}](q)$ and $[conf_{13}](q)$ we know that for LT3 to be enabled in state q requires $|WAITM| > 0$ in state q, and therefore in state q'. Hence $[vbls_{13}](q')$ is true.

13-LT4: Since LT4 increments m, $en(m)$ is true in state q'.

13-LT5: Since LT5 adds an element to m_en, $en(m)$ is true in state q'.

13-LT6: Even though $|LP5|$ becomes positive in state q', by the truth of $[aux_{20}](q)$, and $[conf_{13}](q)$ we know that for LT6 to be enabled in state q requires $|WAITM| = 0$ in state q, and therefore in state q'. Hence $vbls_{13}$ is true in state q'.

13-LT7-1: Since LT7 increments a, $en(a)$ is true in state q'. By the assumptions defining this case, $|MUTEX1| = 0$ in state q, and hence *en*(b) is true in state q. Since *en*(b) is not affected by LT7, *en*(b) is true in state q' as well. $|WAITB2| = 0$ in state q, and is not affected by LT7. For LT7 to be enabled in state q we must have $|LP5| > 0$, and hence $|WAITM| = 0$ in state q, and in state q'. Under these conditions, $[vbls_1](q')$ is a direct consequence of $[vbls_{13}](q)$.

13-LT7-2: Since by the assumptions defining this case, $|MUTEX1| > 0$, we know from $[vbls_{13}](q)$ and $[aux_{17}](q)$ that *en*(b) is false in state q. Since *en*(b) is not affected by LT7, it is false in state q' as well. That $|WAITB2| = 0$ and *en*(a) are true in state q' may be argued as in case 13-LT7-1.

13-LT7-3: Since by the assumptions defining this case, ***procnotcounted*** is false in state q, and since *procnotcounted* is not affected by LT7, it is false in state q' as well. That *en*(b) and *en*(a) are true in state q' may be argued as in case 13-TT7-1.

13-LT8-1: Similar to case 13-LT7-1, except that *en*(a) is true in state q because LT8 adds an element to *a_en*.

13-LT8-2: Similar to case 13-LT7-2, except that *en*(a) is true in state q because LT8 adds an element to *a_en*.

13-LT8-3: Similar to case 13-LT7-3, except that *en*(a) is true in state q because LT8 adds an element to *a_en*.

13-LT9: Transition LT9 does not affect $vbls_{13}$.

References

[BRIN72a] Brinch Hansen, P., "A Comparison of Two Synchronizing Concepts," Acta Informatica 1, (1972), pp. 190-199.

[BRIN72b] Brinch Hansen, P., "Structured Multiprogramming," CACM 15,7 (1972), pp. 574-578.

[BURNS79] Burns, J.E., et al., "Data Requirements for Implementation of N-Process Mutual Exclusion Using a Single Shared Variable," Georgia Institute of Technology Report GIT-ICS-79/02, May 1979.

[COFFM73] Coffman, E.G., and Denning, P.J., Operating System Theory, Prentice Hall, 1973.

[COURT71] Courtois, P.J., F. Heymans, D. Parnas, "Concurrent Control with 'Readers' and 'Writers'," CACM 14, 10 (1971), pp. 667-668.

[COURT72] Courtois, P.J., F. Heymans, D. Parnas, "Comments on 'A Comparison of Two Synchronizing Concepts by P.B. Hansen'," Acta Informatica 1 (1972), pp. 375-376.

[DOEPP76] Doeppner, T.W., "On Abstractions of Parallel Programs," Eighth ACM Symposium on Theory of Computation," 1976, pp. 65-72.

[DIJKS68] Dijkstra, E.W., "Cooperating Sequential Processes," in Programming Languages, F. Genuys (Ed.), Academic Press, 1968, pp. 43-112.

[DIJKS71] Dijkstra, E.W., "Hierarchical Ordering of Sequential Processes," Acta Informatica, 1, (1972), pp. 115-138.

[DIJKS65] Dijkstra, E.W., "Solution of a Problem in Concurrent

Programming Control," CACM 8,9 (1965), p. 569.

[FLOYD67] "Assigning Meanings to Programs," in Mathematical Aspects of Computer Science, American Math. Soc., 1967.

[HABER72] Habermann, A.N., "Synchronization of Communicating Processes," CACM 15, 3 (1972), pp. 171-176.

[HABER75] Habermann, A.N., "Path Expressions," Carnegie-Mellon University, 1975.

[HABER76] Habermann, A.N., "Review of Article by Leon Presser on Multiprogramming Coordination," Computing Reviews 29,788 (April 1976), pp. 150-151.

[HOARE74] Hoare, C.A.R., "Monitors: An Operating System Structuring Concept," CACM 17,10 (1974), pp. 549-557.

[HOLT70] Holt, A., and Commoner, F., "Events and Conditions", Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, June, 1970, pp. 3-52.

[KELLE76] Keller, R.M., "Formal Verification of Parallel Programs," CACM 19,7 (July 1976), pp. 371-384.

[KNUTH66] Knuth, D.E., "Additional Comments on a Problem in Concurrent Programming Control," CACM 9,5 (1966), pp. 321-322.

[KOSAR73] Kosaraju, S.R., "Limitations of Dijkstra's Semaphore Primitives and Petri Nets," Johns Hopkins University Computer Science Report No. 25, 1973.

[KWONG78] Kwong, Y.S., "Livelocks in Parallel Programs," Mc. Master University Technical Report 78-CS-15.

[LAMPO74] Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem," CACM 17,8 (1974), pp. 453-455.

[LIPTO73] Lipton, R.J., "On Synchronization Primitive Systems," PhD thesis, Carnegie-Mellon University, (1973).

[MILLE77] Miller, R.E. and C.K. Yap, "Formal Specification and Analysis of Loosely Connected Processes," IBM Research Report RC6716, (1977).

[OWICK75] Owicki, S.S., "Axiomatic Proof Techniques for Parallel Programs," PhD thesis, Cornell University, (1975).

[PRESS75] Presser, Leon, "Multiprogramming Coordination," Computing Surveys 7,1 (March 1975), pp. 21-44.

[SHAW74] Shaw, A.C., The Logical Design of Operating Systems, Prentice Hall, 1974, p. 78.

[WODON72] Wodon, P., "Still Another Tool for Synchronizing Cooperating Processes," Carnegie-Mellon University Report, (1972).