An Investigation of Models of Concurrent Programs

David O'Hallaron, Paul F. Reynolds, Jr.

Computer Science Report No. TR-86-05
March 12, 1986

Abstract

Reynolds has proposed an interesting new approach to verifying concurrent programs, called *static incremental deadlock detection*, where deadlock potential is automatically detected *during* the interactive construction of a concurrent program, *before* the program is ever run. In order to perform effective static incremental deadlock detection, we will require accurate models that can be analyzed efficiently. Towards this end, we investigate three existing classes of models: algebraic models, geometric models, and Petri net models. We report new results related to the accuracy of algebraic and geometric models, and we report new results on the efficient analysis of Petri net models. Finally, we use our results to derive a technique for performing effective static incremental deadlock detection on a subclass of concurrent programs.

Table of Contents

List of Definitions

List of Lemmas

List of Theorems

List of Figures

List of Symbols

General

Σ	iterative sum
$\prod$	iterative product
$\forall$	for all
$\exists$	there exists
$\nexists$	there does not exist
$\wedge$	logical conjunction
$\vee$	logical disjunction
$\Rightarrow$	logical implication
$\Leftrightarrow$	logical equivalence
$\cup$	set union
$\cap$	set intersection
$\in$	is a member of
$\notin$	is not a member of
$\subseteq$	is a subset of
$\not\subseteq$	is not a subset of
$\|A\|$	cardinality of set A

Semaphore Programs

Algebraic Models

Geometric Models

Petri Net Models

Chapter 3

Chapter 7

CHAPTER 1

Introduction

In [REYN79], Reynolds proposes an interesting new approach to verifying concurrent programs, where deadlock potential is detected *during* the interactive construction of a concurrent program, *before* the program is ever run. The construction of a program is viewed as a sequence of discrete steps. After each step an analysis tool automatically analyzes the partially completed program for deadlock potential. This new approach to program verification is called *static incremental deadlock detection* and is motivated by the desire to simplify the construction of concurrent programs by detecting errors as early as possible in the development cycle.

To date, no results for building analysis tools that perform static incremental deadlock detection have appeared in the literature. However, we can deduce some attributes required of any effective tool that performs static incremental deadlock detection. First, because deadlock potential is detected before a program is ever run, the tool must analyze a *model* of the program under construction. Second, because the program is constructed interactively, we insist that the analysis of the model be *efficient*. Granted the need for efficiency is always acute, but it is especially so in an interactive environment. Finally, we require that the model be *accurate*, that it accurately capture the deadlock potential of the program it represents, with no possibility of false reports of deadlock freedom or deadlock potential. In sum, we seek analysis tools that perform efficient and accurate static

incremental deadlock detection on models of concurrent programs.

Towards that end, we investigate models of concurrent programs. In particular, we investigate three existing classes of models: algebraic models, geometric models, and Petri net models. We report new results related to the accuracy of algebraic and geometric models, and we report new results on the efficient analysis of Petri net models. Finally, we use the knowledge we have gained to derive a technique for performing effective static incremental deadlock detection on a restricted class of concurrent programs.

As a common vehicle for illustrating our results, we have chosen a class of cyclic non-terminating semaphore programs[DIJK68] where each process consists of straightline P and V operations. We have chosen these programs because they are powerful enough to solve many interesting synchronization problems, and because they are simple enough to be represented by a wide variety of models.

1.1. Models

A model of a concurrent system is a static mathematical representation that captures certain dynamic properties of the system. The model can be analyzed to decide whether the system will exhibit these properties when it is run. Different kinds of models can be compared according to their *modeling power* and their *decision power* [KELL77]. Modeling power is a measure of the range of systems that can be represented by a particular kind of model. Decision power is a measure of the range of dynamic properties that are captured by a particular kind of model.

We are concerned here with models that capture synchronization properties of a concurrent system. Two synchronization properties that have received attention

in the literature are reachability and deadlock potential.

If a model of a system captures the property of reachability, then we can decide from the model whether the synchronization constraints of the system allow the system to maneuver itself into some arbitrary "state". The notion of reachability was first identified for vector addition systems in [KARP69].

Reachability has been studied extensively in the Petri net literature. Representative works include [KARP69], [HOLT70], [COMM71], [GENR73], [CRES75], [HACK76], [LIPT76.], [ARAK77], [LAND78], [YAMA81], [MAYR81], [KOSA82], [MAYR84], and [YAMA84].

If a model of a system captures the property of deadlock potential, then we can decide from the model whether the system can ever maneuver itself into a state where one or more processes can never make further progress. Deadlock potential was first identified for semaphore programs in [DIJK68].

Models that capture deadlock potential have been treated in the communications, operating systems, and Petri net literature. Models studied in the communications literature include communicating finite state machine models [ZAFI80], [SIDH82], [YU82], [GOUD83], and [CHOW84], graph models [MERL80], [GUNT81], [WIMM84], and [BLAZ84], and temporal logic models [HAIL83]. Models studied in the Petri net literature include various subclasses of Petri nets [HOLT70], [COMM72], [HACK72], [HACK76], [LIEN76], [HERZ77], [MEMM78], [HERZ79], [GRIE79], and [DATT84]. Models studied in the operating systems literature include resource ordering models [HAVE68], graph models [HOLT72], semaphore invariant models [HABE72], algebraic models [CLAR80, OWIC76], coincidence network models [HANS81], temporal logic models [OWIC82], symbolic execution models [MANC83], flow graph

models [TAYL83], and geometric models [CARS84].

1.2. Accurate Algebraic and Geometric Models

An algebraic model[CLAR80, OWIC76] is a predicate that can be evaluated to determine the potential for total deadlock, a condition where every process is blocked forever. Unfortunately, as defined in the literature, the algebraic model does not capture partial deadlock, a condition where a subset of processes is blocked forever.

In this work we shall describe a new technique using predicate transformers for augmenting the decision power of algebraic models so that the potential for partial deadlock can be decided from the model.

A geometric model[CARS84] is based on a formalization of the familiar notion of progress graphs[COFF71]. To build a geometric model of a cyclic semaphore program, we must first decide how many finite loops of each process to "unfold" in order for the model to accurately capture deadlock potential. This problem, called the *finite models problem*, is open for general semaphore programs, although Carson has solved the problem for the subclass of reusable resource semaphore programs.

In this work, we solve the finite models problem for a useful subclass of consumable resource programs.

1.3. Efficient Analysis of Petri Net Models

Reachability for Petri nets is decidable[MAYR81, MAYR84]. Since deciding liveness(deadlock-freedom) is reducible to deciding reachability[HACK76], liveness is

decidable as well. The worst-case complexity of the decision algorithm for reachability is unknown[MAYR84], although it is known to be at least exponential[LIPT76]. Furthermore, it is not known if the algorithm is implementable. Thus, the search for subclasses of Petri nets with simpler algorithms for deciding reachability and liveness remains an important area of research.

In this work, we identify a new subclass of Petri nets for which reachability can be decided by an enumeration of a finite number of states, even though the number of reachable states is potentially infinite. The decision algorithm requires exponential time in the worst case, but it is simple to understand and implement.

Also, we identify a new subclass of Petri nets for which liveness can be decided in polynomial time.

Finally, we use this Petri net result to develop a technique for performing efficient and accurate static incremental deadlock detection on a subclass of semaphore programs.

1.4. Structure of the Thesis

In Chapter 2, we give formal definitions for semaphore programs and for algebraic, geometric, and Petri net models.

In Chapter 3, we describe a new technique using predicate transformers for augmenting the decision power of algebraic models so that both freedom from partial and total deadlock can be decided from the model.

In Chapter 4, we use theory from algebraic models and Petri net models to solve the finite models problem for a subclass of consumable resource semaphore programs.

In Chapter 5, we derive a reachability theorem for general Petri nets. We use this theorem to show that for a certain subclass of Petri nets, called *bounded inverse* nets, reachability can be decided by a simple enumeration of states, even though the number of reachable states is potentially infinite. Then we show how this result can be used to analyze semaphore programs for reachability.

In Chapter 6, we derive necessary and sufficient conditions for the liveness(deadlock-freedom) of a new subclass of Petri nets called *SM* Petri nets. We use these results to add *SM* nets to the list of Petri net subclasses for which the *deadlock–trap* property of [HACK72] is a necessary and sufficient condition for liveness. We then show how our results can be used to analyze a subclass of semaphore programs for deadlock-freedom.

Finally, in Chapter 7, we use the results of Chapter 6 to show how static incremental deadlock detection might be performed on a subclass of semaphore programs.

CHAPTER 2

Background

In this chapter we discuss concepts and notations for semaphore programs and their algebraic, geometric, and Petri net models. For each model, we give basic definitions and discuss the model's construction, analysis, modeling power, and decision power.

2.1. Semaphore Programs

A *semaphore program*, C, is a collection of N cyclic processes

$$\textbf{var } L\textbf{: semaphore; cobegin } P_1 \,//\, P_2 \,//\, \cdots \,//\, P_N \textbf{ coend}$$

where L is a finite list of *semaphores* and their initial nonnegative values. Each P_i is a process of the form

$$P_i : \textbf{cycle } S_i^1; S_i^2; \cdots ; S_i^{k_i} \textbf{ endcycle.}$$

Each S_i^k is a statement of the form

$$\textbf{when } \neg b \textbf{ do } A,$$

where b is a *blocking condition* and A is an indivisible sequential statement. If S_i^k *requests* semaphore σ, then b is the predicate $(\sigma = 0)$, A is the sequential statement $\sigma \leftarrow \sigma - 1$, and S_i^k is denoted $P(\sigma)$ in the program text. If S_i^k *releases* semaphore σ, then b is the predicate **false**, A is the sequential statement $\sigma \leftarrow \sigma + 1$, and S_i^k is denoted $V(\sigma)$ in the program text.

As in [CLAR80], we say that a *program state* $\tau \in T$ is an ordered list $(pc_1, \ldots, pc_N; s)$, where pc_i is the *program counter* for process i with $1 \leqslant pc_i \leqslant k_i$, and where s is the *program store* representing the values of the semaphores in state τ. The *initial state* τ_0 has the form $(1, \ldots, 1; s_0)$ where s_0 is the store reflecting the initial values of the semaphores. We say that $\sigma(s)$ is the value of semaphore σ in store s, $b(s)$ is the truth value of the blocking condition in store s, and $A(s)$ is the new store resulting when sequential statement A is executed in store s. If $\sigma(s) < 0$ for any semaphore σ, then τ is *infeasible*; otherwise τ is *feasible*.

A statement

$$S_i^k \equiv \textbf{when } \neg b \textbf{ do } A,$$

is *executable* in state $\tau = (pc_1, \ldots, pc_N; s)$ if $pc_i = k$ and $\neg b(s) = \textbf{true}$. *Executing* an executable statement S_i^k in τ leads to a new state $\tau' = (pc_1, \ldots, pc_i', \ldots, pc_N; A(s))$, where

$$pc_i' = \begin{cases} pc_i + 1 & \text{if } pc_i < k_i \\ 1 & \text{otherwise} \end{cases}$$

We let $\tau \xrightarrow{S} \tau'$ denote that the execution of a statement S that is executable in τ leads to state τ'.

The notion of execution can be extended to finite sequences of statements. The empty sequence λ is executable in every state τ. For a sequence of statements w and a statement S, the sequence Sw is executable in τ if S is executable in τ, $\tau \xrightarrow{S} \tau'$, and the sequence w is executable in τ'. We let $\tau \xrightarrow{w} \tau'$ denote that w is executable in τ and that executing w in τ leads to state τ'.

A state τ' is *reachable* from state τ if there exists a sequence w executable in τ such that $\tau \xrightarrow{w} \tau'$. If τ is reachable from τ_0, we simply say that τ is reachable. Note that reachable states are feasible, while feasible states are not necessarily reachable.

A *computation* is a finite sequence of states $\tau\tau = \tau_0\tau_1 \cdots \tau_k$ such that for each τ_i and τ_{i+1} in $\tau\tau$, $0 \leqslant i < k$, there is some statement S such that $\tau_i \xrightarrow{S} \tau_{i+1}$. We say that τ_i is an *immediate predecessor* of τ_{i+1} and that τ_{i+1} is an *immediate successor* of τ_i.

A statement S is *dead* in τ if S is executable in no states reachable from τ. A statement S is *live* in τ if for all τ' reachable from τ, there exists a sequence of statements w such that $\tau' \xrightarrow{w} \tau''$ and S is executable in τ''. A state τ is live if every statement is live in τ. If the initial state τ_0 of program C is live, then we say that C is deadlock-free.

Process i is *blocked* at statement

$$S_i^k \equiv \textbf{when } \neg b \textbf{ do } A\,,$$

in state $\tau = (pc_1, \ldots, pc_N; s)$ if $pc_i = k$ and $b(s) = \textbf{true}$. Process i is *deadlocked* at statement S_i^k in state $\tau = (pc_1, \ldots, pc_N; s)$ if $pc_i = k$ and S_i^k is dead in τ. We say that τ is a deadlock state if at least one process is deadlocked in τ. If τ is a deadlock state and $k < N$ processes are deadlocked in τ, then τ is a *partial* deadlock state; otherwise τ is a *total* deadlock state.

2.1.1. Examples of Semaphore Programs

Semaphore programs are sufficiently powerful to solve many of the classical synchronization and resource allocation problems, including mutual exclusion, producer/consumer, dining philosophers, and message passing. We give examples of semaphore programs that solve the mutual exclusion problem, the producer/consumer problem, and a simple message passing problem.

Figure 2-1 is a semaphore program with two processes that enforces mutually exclusive access to some shared resource. Access to the resource is controlled by semaphore m. Since m is initially unity, only one process is allowed access to the resource at any time.

Figure 2-2 is a two process producer/consumer system operating on a bounded buffer with two slots. The producer process repeatedly waits for an empty slot in the buffer, get exclusive access to the buffer, and fills the empty slot with an

```
var m = 1: semaphore;

cobegin
        1: cycle
                P(m)        — request exclusive access to the resource
                V(m)        — release exclusive access
           endcycle
           //
        2: cycle
                P(m)        — request exclusive access to the resource
                V(m)        — release exclusive access
           endcycle
coend
```

Figure 2-1: *A Mutual Exclusion Program.*

```
var m = 1, s = 2, e = 0: semaphore

cobegin
        1: cycle                    — producer process
                P(s)                — request a slot
                P(m)                — request exclusive access to the buffer
                V(m)                — release exclusive access
                V(e)                — produce an element
        endcycle
        //
        2: cycle                    — consumer process
                P(e)                — request an element
                P(m)                — request exclusive access to the buffer
                V(m)                — release exclusive access
                V(s)                — produce a slot
        endcycle
coend
```

Figure 2-2: *A Producer/Consumer Program.*

element. The consumer process repeatedly waits for a new element to appear in the buffer, gets exclusive access to the buffer, and removes the element from its slot. In essence, the producer process produces elements and consumes slots, while the consumer process produces slots and consumes elements.

Our final example is the simple message-passing system in Figure 2-3, where a process sending a message continues to the next statement without awaiting a response from the receiver. The program has eight reachable states:

$$\tau_0 = (1,1;a=0,b=0,c=0,d=0) \quad \tau_4 = (3,1;a=1,b=1,c=0,d=0)$$

$$\tau_1 = (2,1;a=1,b=0,c=0,d=0) \quad \tau_5 = (3,2;a=1,b=2,c=0,d=0)$$

$$\tau_2 = (1,2;a=0,b=1,c=0,d=0) \quad \tau_6 = (2,3;a=0,b=1,c=0,d=0)$$

$$\tau_3 = (2,2;a=1,b=1,c=0,d=0) \quad \tau_7 = (3,3;a=0,b=2,c=0,d=0)$$

State τ_0 is the initial state where neither process has executed a statement. State τ_1 is the state where process 1 is ready to execute statement S_1^2, and process 2 is ready to execute statement S_2^1.

There are several problems with the program in Figure 2-3. State τ_7 is a total deadlock state; each process is blocked forever. States τ_4, τ_5, and τ_6 are partial deadlock states; one process is deadlocked while the other can still make progress. Finally, since both processes release semaphore b and neither process requests b, the program has an infinite number of feasible states; only eight of these are reachable because of the inevitable total deadlock.

Admittedly, these errors are easy to detect in a program this small. Nonetheless, they will help us illustrate the decision power of algebraic, geometric, and Petri net models of semaphore programs.

2.1.2. Subclasses of Semaphore Programs

The class of semaphore programs can be divided into three disjoint subclasses[HOLT72]: reusable resource programs, consumable resource programs, and general resource programs. In a reusable resource program, a semaphore is requested and released symmetrically by the same process. In a consumable resource program, a semaphore is requested and released by different processes. All other semaphore programs are called general resource programs. Figure 2-1 is an example of a reusable resource program. Figure 2-3 is an example of a consumable resource program. Figure 2-2 is an example of a general resource program.

```
var a,b,c,d = 0: semaphore

cobegin
        1: cycle
                V(a)        — send message a
                V(b)        — send message b
                P(d)        — wait for message d
                V(c)        — wait for message c
        endcycle
        //
        2: cycle
                V(b)        — send message b
                P(a)        — wait for message a
                P(c)        — wait for message c
                V(d)        — send message d
        endcycle
coend
```

Figure 2-3: *A Simple Message Passing Program.*

2.2. Algebraic Models

An algebraic model of a semaphore program is a predicate called the *total deadlock predicate*. If the total deadlock predicate is satisfiable, then there exists a feasible total deadlock state. The model was developed by Owicki and Gries in [OWIC76] and is based on the notion of *resource invariants*. The resource invariants for semaphore programs (*semaphore invariants*) were identified by Habermann in [HABE72]. A technique for synthesizing the deadlock predicate from the program text was developed by Clarke in [CLAR80] and later generalized by Carson in [CARS84].

2.2.1. Definitions

The deadlock predicate consists of semaphore invariants, auxiliary variables, preconditions, and blocking conditions.

Let σ be a semaphore initialized to σ_0. Let σ_i^P and σ_i^V be auxiliary variables, initially zero, such that σ_i^P is incremented when process i acquires σ(completes a P operation on σ), and such that σ_i^V is incremented when process i releases σ(completes a V operation on σ). Then the semaphore invariant for σ is defined by

$$I_\sigma \equiv \sigma = \sigma_0 - \sum_{i=1}^{N} \sigma_i^P + \sum_{i=1}^{N} \sigma_i^V \wedge \sigma \geqslant 0 \wedge \forall i : \sigma_i^P \geqslant 0 \wedge \forall i : \sigma_i^V \geqslant 0.$$

A state is uniquely defined by values of all of the auxiliary variables for all semaphores. If the values of the auxiliary variables do not satisfy the semaphore invariants of every semaphore, then the associated state is infeasible.

Associated with each process i is a set of auxiliary variables V_i, where $\sigma_i^P \in V_i$ if and only if process i requests σ, and where $\sigma_i^V \in V_i$ if and only if process i releases σ.

The *precondition* of the *jth* statement in the *ith* process, $pre(S_i^k)$, describes the relationships that exist among the auxiliary variables in V_i just before statement S_i^k is executed. We shall see later in this section how the preconditions are constructed from the text of the program.

Associated with each statement S_i^k is a *blocking condition*, b_i^k, that describes the conditions under which S_i^k is blocked. If S_i^k is $V(\sigma)$ then b_i^k is simply the predicate **false**. If S_i^k is $P(\sigma)$ then b_i^k is the predicate $(\sigma = 0)$.

Let S be the number of semaphores used in the program and let k_i be the number of statements in process i. Then the total deadlock predicate is defined by

$$D = \bigwedge_{i=1}^{N} \left\{ \bigvee_{j=1}^{k_i} pre(S_i^j) \wedge b_i^j \right\} \wedge \left\{ \bigwedge_{\zeta=1}^{S} I_\zeta \right\}.$$

The first term of the total deadlock predicate is satisfied by those states where each process is blocked. The conjunction of the semaphore invariants eliminates the infeasible states.

2.2.2. Constructing the Model

The procedure for constructing the deadlock predicate from the text of a program was developed by Clarke in [CLAR80] and later generalized by Carson in [CARS84]. Consider the semaphore program in Figure 2-3. To generate the deadlock predicate for this program, we first associate auxiliary variables with the statements of each process. This step is shown in Figure 2-4.

Next we generate the statement preconditions. Recall that V_i is the set of auxiliary variables associated with process i. Let v_i^k be the kth auxiliary variable in V_i and let m_i^k be the number of times that v_i^k is incremented in one cycle of process i. Let d_i^j be the auxiliary variable associated with statement S_i^j. Then the preconditions are defined recursively by

$$pre(S_i^1) \equiv post(S_i^{k_i}) \equiv \{(\prod_{k \neq 1} m_i^k)v_i^1 = (\prod_{k \neq 2} m_i^k)v_i^2 = \cdots = (\prod_{k \neq |V_i|} m_i^k)v_i^{|V_i|}\}$$

and

$$post(S_i^j) \equiv pre(S_i^j) \text{ with } d_i^j \text{ replaced by } d_i^j - 1.$$

```
a,b,c,d = 0: semaphore

{a_1^V = b_1^V = d_1^P = c_1^V = b_2^V = a_2^P = c_2^P = d_2^V = 0}
cobegin
        1: cycle
                V(a) {a_1^V ← a_1^V + 1}
                V(b) {b_1^V ← b_1^V + 1}
                P(d) {d_1^P ← d_1^P + 1}
                V(c) {c_1^V ← c_1^V + 1}
        endcycle
        //
        2: cycle
                V(b) {b_2^V ← b_2^V + 1}
                P(a) {a_2^P ← a_2^P + 1}
                P(c) {c_2^P ← c_2^P + 1}
                V(d) {d_2^V ← d_2^V + 1}
        endcycle
coend
```

Figure 2-4: *Semaphore Program Annotated With Auxiliary Variables.*

Since each semaphore in our example appears exactly once per process, the initial precondition for process 1 is simply

$$pre(S_1^1) \equiv \{a_1^V = b_1^V = d_1^P = c_1^V\}$$

and the initial precondition for process 2 is

$$pre(S_2^1) \equiv \{b_2^V = a_2^P = c_2^P = d_2^V\}.$$

If we then apply the recursive definition, we get the complete set of preconditions shown in Figure 2-5.

Next, we construct the semaphore invariant for each semaphore that appears in the program.

$$I_a \equiv a = a_1^V - a_2^P \wedge a \geqslant 0 \wedge a_1^V \geqslant 0 \wedge a_2^P \geqslant 0$$

$$I_b \equiv b = b_1^V + b_2^V \wedge b \geqslant 0 \wedge b_2^V \geqslant 0 \wedge b_1^V \geqslant 0$$

$$I_c \equiv c = c_1^V - c_2^P \wedge c \geqslant 0 \wedge c_1^V \geqslant 0 \wedge c_2^P \geqslant 0$$

$$I_d \equiv d = d_2^V - d_1^P \wedge d \geqslant 0 \wedge d_2^V \geqslant 0 \wedge d_1^P \geqslant 0$$

Combining the precondition and the blocking condition associated with statement S_i^j, we get a blocking precondition of the form

$$(pre(S_i^j) \wedge \sigma = 0)$$

if statement S_i^j requests σ, or

$$(pre(S_i^j) \wedge \textbf{false})$$

if statement S_i^j releases σ. For example, the blocking precondition associated with the third statement of the first process in our example is

a,b,c,d = 0: **semaphore**

$\{a_1^V = b_1^V = d_1^P = c_1^V = b_2^V = a_2^P = c_2^P = d_2^V = 0\}$
cobegin
1: **cycle**
$\{a_1^V = b_1^V = d_1^P = c_1^V\}$
V(a) $\{a_1^V \leftarrow a_1^V + 1\}$
$\{a_1^V - 1 = b_1^V = d_1^P = c_1^V\}$
V(b) $\{b_1^V \leftarrow b_1^V + 1\}$
$\{a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V\}$
P(d) $\{d_1^P \leftarrow d_1^P + 1\}$
$\{a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V\}$
V(c) $\{c_1^V \leftarrow c_1^V + 1\}$
endcycle
//
2: **cycle**
$\{b_2^V = a_2^P = c_2^P = d_2^V\}$
V(b) $\{b_2^V \leftarrow b_2^V + 1\}$
$\{b_2^V - 1 = a_2^P = c_2^P = d_2^V\}$
P(a) $\{a_2^P \leftarrow a_2^P + 1\}$
$\{b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V\}$
P(c) $\{c_2^P \leftarrow c_2^P + 1\}$
$\{b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V\}$
V(d) $\{d_2^V \leftarrow d_2^V + 1\}$
endcycle
coend

Figure 2-5: *Semaphore Program Annotated With Preconditions.*

$$(a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V \wedge d = 0).$$

We construct the total deadlock predicate by simply combining the disjuncts and the semaphore invariants with the correct logical connectives according to the definition. The total deadlock predicate for the example program is shown in Figure 2-6.

2.2.3. Modeling Power and Complexity

As defined above, the total deadlock predicate can model all semaphore programs. Furthermore, Clarke has developed a generalized semaphore invariant[CLAR80] that can be used to construct a deadlock predicate for a larger class of programs that synchronize with conditional critical regions.

One of the strengths of the deadlock predicate is the simplicity of its construction, requiring time and space polynomial in the number of processes. However, this simplicity does not come without costs, as we shall see.

2.2.4. Analyzing the Model

To analyze an algebraic model of a semaphore program we must determine what (if any) nonnegative values of the auxiliary variables satisfy the deadlock predicate. We might do this by selecting a set of N disjuncts (blocking preconditions) from the deadlock predicate (one disjunct from each process), conjoining these with the semaphore invariants, and determining by some means if there exist any values of the auxiliary values that satisfy the conjunction. If so, then the program has a feasible total deadlock state. If not, then we select another set of disjuncts and repeat the process until we find a solution or until all possible sets of dis-

$$
\begin{aligned}
D = \; & [\,(a_1^V = b_1^V = d_1^P = c_1^V \wedge \textbf{false}) \\
& \vee (a_1^V - 1 = b_1^V = d_1^P = c_1^V \wedge b = 0) \\
& \vee (a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V \wedge d = 0) \\
& \vee (a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V \wedge \textbf{false})] \\
\wedge \\
& [\,(b_2^V = a_2^P = c_2^P = d_2^V \wedge \textbf{false}) \\
& \vee (b_2^V - 1 = a_2^P = c_2^P = d_2^V \wedge a = 0) \\
& \vee (b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V \wedge c = 0) \\
& \vee (b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V \wedge \textbf{false})] \\
& \wedge (a = a_1^V - a_2^P \wedge a \geqslant 0 \wedge a_1^V \geqslant 0 \wedge a_2^P \geqslant 0) \\
& \wedge (b = b_1^V + b_2^V \wedge b \geqslant 0 \wedge b_2^V \geqslant 0 \wedge b_1^V \geqslant 0) \\
& \wedge (c = c_1^V - c_2^P \wedge c \geqslant 0 \wedge c_1^V \geqslant 0 \wedge c_2^P \geqslant 0) \\
& \wedge (d = d_2^V - d_1^P \wedge d \geqslant 0 \wedge d_2^V \geqslant 0 \wedge d_1^P \geqslant 0)
\end{aligned}
$$

Figure 2-6: *Deadlock Predicate.*

juncts have been considered.

Suppose we select the disjuncts corresponding to the third statement of process 1 and the second statement of process 2 of our example and conjoin these with the semaphore invariants.

$$(a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V \wedge d = 0)$$
$$\wedge (b_2^V - 1 = a_2^P = c_2^P = d_2^V \wedge a = 0)$$
$$\wedge (a = a_1^V - a_2^P \wedge a \geqslant 0 \wedge a_1^V \geqslant 0 \wedge a_2^P \geqslant 0)$$
$$\wedge (b = b_1^V + b_2^V \wedge b \geqslant 0 \wedge b_1^V \geqslant 0 \wedge b_2^V \geqslant 0)$$
$$\wedge (c = c_1^V - c_2^P \wedge c \geqslant 0 \wedge c_1^V \geqslant 0 \wedge c_2^P \geqslant 0)$$
$$\wedge (d = d_2^V - d_1^P \wedge d \geqslant 0 \wedge d_2^V \geqslant 0 \wedge d_1^P \geqslant 0).$$

Substituting into the invariant for semaphore a we have

$$\begin{aligned} 0 &= a_1^V - a_2^P \\ &= (d_1^P + 1) - a_2^P \\ &= (d_1^P + 1) - d_2^V \\ &= d_1^P - d_2^V + 1 \end{aligned}$$

which implies that $d_2^V > d_1^P$, which implies that $d_2^V - d_1^P > 0$, which implies that $d > 0$, which contradicts the blocking condition $(d = 0)$. Thus this combination of disjuncts is unsatisfiable.

Now suppose we select the disjuncts associated with the third statement of each process. Combining these with the semaphore invariants we get the following equation:

$$(a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V \wedge b = 0)$$
$$\wedge (b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V \wedge a = 0)$$

$$\wedge (a = a_1^V - a_2^P \wedge a \geqslant 0 \wedge a_1^V \geqslant 0 \wedge a_2^P \geqslant 0)$$
$$\wedge (b = b_1^V + b_2^V \wedge b \geqslant 0 \wedge b_2^V \geqslant 0 \wedge b_1^V \geqslant 0)$$
$$\wedge (c = c_1^V - c_2^P \wedge c \geqslant 0 \wedge c_1^V \geqslant 0 \wedge c_2^P \geqslant 0)$$
$$\wedge (d = d_2^V - d_1^P \wedge d \geqslant 0 \wedge d_2^V \geqslant 0 \wedge d_1^P \geqslant 0).$$

There are an infinite number of solutions to this predicate, each solution corresponding to a feasible total deadlock state. One such solution is

$$a_1^V = 1 \quad a_2^P = 1$$
$$b_1^V = 1 \quad b_2^V = 1$$
$$c_1^V = 0 \quad c_2^P = 0$$
$$d_2^V = 0 \quad d_1^P = 0$$

which corresponds to the total deadlock state τ_7. Inspection shows that τ_7 is reachable, hence the program is not deadlock-free.

2.2.5. Decision Power and Complexity

The total deadlock predicate is satisfied by all feasible total deadlock states. Thus satisfiability of the deadlock predicate is a necessary condition for total deadlock. Unfortunately, there are some deficiencies in the decision power of the deadlock predicate.

First, satisfiability of the total deadlock predicate is not a sufficient condition for total deadlock because of the possibility of feasible yet unreachable deadlock states called trap states[CLAR80]. Consider our example program in Figure 2-3 and its deadlock predicate in Figure 2-6. The predicate is also satisfied by

$$a_1^V = 2 \quad a_2^P = 2$$
$$b_1^V = 2 \quad b_2^V = 2$$
$$c_1^V = 1 \quad c_2^P = 1$$
$$d_2^V = 1 \quad d_1^P = 1$$

which corresponds to the total deadlock state $(3,3;a=0,b=4,c=0,d=0)$. However, it is clear that this state is not reachable. The deadlock predicate is not sufficiently powerful to decide the reachability of a total deadlock state, and thus its satisfiability is not a sufficient condition for total deadlock.

Second, the deadlock predicate is of no help in analyzing a program for partial deadlock[CARS84.]. Consider the partial deadlock state τ_4. When this state is reached during the first cycle of each process the values of the auxiliary variables

$$a_1^V = 1 \quad a_2^P = 0$$
$$b_1^V = 1 \quad b_2^V = 0$$
$$c_1^V = 0 \quad c_2^P = 0$$
$$d_2^V = 0 \quad d_1^P = 0$$

do not satisfy the deadlock predicate. Thus satisfiability of the deadlock predicate is not even a necessary condition for partial deadlock.

We also note that solving the deadlock predicate requires worst-case time exponential in the number of processes[CARS84].

2.3. Geometric Models

A geometric model[CARS84] of a semaphore program with N processes is a potentially infinite set of potentially infinite N dimensional rectangular structures (known as forbidden regions) scattered about a continuous potentially infinite N dimensional space. The model is analyzed for such properties as total deadlock, partial deadlock, and reachability by investigating the spatial relationships of the forbidden regions.

In this section we review Carson's recent work and we discuss how geometric models are used to model and analyze semaphore programs.

2.3.1. Definitions

The geometric model is a formalization of the familiar notion of progress graphs [COFF71], where an instance of a state is modeled as a point in a continuous N dimensional space. The *ith* time axis is labeled with synchronization events (P and V operations) in the order in which they executed by process i. For convenience synchronization events are assigned integer times starting from unity. Thus, for two processes, the point (0,0) represents the initial state where neither process has completed a synchronization event and the point (3,1) represents the state where processes 1 and 2 have completed their third and first synchronization events, respectively.

Sets of states with like synchronization properties are modeled by collections of points called *regions*. Paraphrasing from [CARS84], a region is an N dimensional rectangular structure which is bounded by $2N$, $N-1$ dimensional bounded hyperplanes. The location and direction of each hyperplane are defined by the equation

$x_i = k$ where x_i is the *ith* coordinate axis and k is some constant. The bounds of each hyperplane are defined by its intersections with the other hyperplanes comprising the region. Each hyperplane is parallel to exactly one other hyperplane within a region, and is orthogonal to all the others. Further, each hyperplane is parallel to all coordinate axes except one (the one that defines the hyperplane).

The size and location of a region are uniquely identified by two points: the *vertex* and the *extent*. The vertex is the point on the region closest to the origin. The extent is the point on the region furthest from the origin. The N hyperplanes defined by the vertex are called *vertex hyperplanes*. The N hyperplanes defined by the extent are called *extent hyperplanes*.

A point $P(p_1, \ldots, p_N)$ is *contained* in a region defined by vertex $V(v_1, \ldots, v_N)$ and extent $E(e_1, \ldots, e_N)$ if

$$\forall_{i \in \{1 \ldots N\}} : \; v_i \leqslant p_i < e_i$$

Two regions defined by vertex $V^1(v_1^1, \ldots, v_N^1)$ and extent $E^1(e_1^1, \ldots, e_N^1)$, and vertex $V^2(v_1^2, \ldots, v_N^2)$ extent $E^2(e_1^2, \ldots, e_N^2)$ *intersect* if and only if

$$\forall_{i \in \{1 \ldots N\}} : \; \max(v_i^1, v_i^2) < \min(e_i^1, e_i^2).$$

If the regions defined by (V^1, E^1) and $V^2, E^2)$ intersect, then the vertex of the region of intersection is a point $V(v_1, \ldots, v_N)$ such that

$$\forall_{i \in \{1 \ldots N\}} : \; v_i = \max(v_i^1, v_i^2)$$

and the extent of the region of intersection is a point $E(e_1, \ldots, e_N)$ such that

$$\forall_{i \in \{1 \ldots N\}} : \; e_i = \min(e_i^1, e_i^2)$$

Regions of interest include *forbidden* regions, *nearness* regions, *deadlock* regions, and *unsafe* regions.

Forbidden regions model states that violate the constraints imposed by the the underlying synchronization primitives. For semaphore programs, points within forbidden regions model states that do not satisfy the conjunction of the semaphore invariants.

Definition 2-1: Forbidden Region

A forbidden region is a triple (σ,V,E), where σ is a semaphore associated with the region, V is the vertex of the region, and E is the extent of the region.

□

Carson has developed a technique for generating the vertices of forbidden regions from the source text of semaphore programs; the technique is based on the notion of *deficits*. Let S_i^j be the *jth* statement executed by the *ith* process of a semaphore program.

Definition 2-2: Deficit

The deficit for semaphore σ at S_i^j,the *jth* synchronization event on the *ith* coordinate axis is denoted by $d(i,j,\sigma)$ and defined by

$$d(i,0,\sigma) = 0$$

$$d(i,j,\sigma) = d(i,j-1,\sigma) + \begin{cases} 1 & \text{if } S_i^j \textit{ requests } \sigma \\ -1 & \text{if } S_i^j \textit{ releases } \sigma \\ 0 & \textit{otherwise} \end{cases}$$

□

Definition 2-3: Deficit at a Point

The deficit at a point P for semaphore σ with initial value σ_0 is denoted by $D(P,\sigma)$ and defined by

$$D(P,\sigma) = -\sigma_0 + \sum_{i=1}^{N} d(i,p_i,\sigma).$$

□

The deficit at a point P for a semaphore σ is simply minus the value of σ at P. A point P is feasible if and only if $D(P,\sigma) \leqslant 0$ for all σ.

Let $use[\sigma]$ be the set of processes that either request or release semaphore σ. Let $stmt(p_i)$ be the statement associated with the ith coordinate at point $P(p_1, \ldots, p_N)$. Possible values for $stmt(p_i)$ are $P(\sigma)$, $V(\sigma)$, and I (the initial statement at the origin).

Definition 2-4: Vertex of a Forbidden Region (σ,V,E)

The vertex of a forbidden region for semaphore σ is a point $V(v_1, \ldots, v_N)$ such that

$$\exists_{i \in use[\sigma]} : stmt(v_i) = P(\sigma),$$

$$\forall_{i \in use[\sigma]} : stmt(v_i) \in \left\{P(\sigma),V(\sigma),I\right\},$$

$$\forall_{i \notin use[\sigma]} : v_i = 0, \text{ and}$$

$$D(V,\sigma) = 1.$$

□

Definition 2-5: Extent of a Forbidden Region (σ,V,E)

The extent of a forbidden region for semaphore σ, whose vertex is $V(v_1, \ldots, v_N)$, is the point $E(e_1, \ldots, e_N)$ such that

$$\forall_{i \in \{1 \ldots N\}} : d(i,e_i,\sigma) = d(i,v_i,\sigma) - 1 \wedge e_i > p_i$$

$$\wedge \nexists_{q_i} : d(i,q_i,\sigma) = d(i,v_i,\sigma) - 1 \wedge v_i < q_i < e_i .$$

If there exists no e_i satisfying the above, then $e_i = \infty$.

□

Nearness regions model states where the progress of a single process is blocked by a vertex hyperplane of a forbidden region.

Definition 2-6: Nearness Region

A nearness region is a triple (d,V,E), where d is the direction in which the system is blocked within the region, V is the vertex of the region, and E is the extent of the region.

□

Up to N nearness regions can be generated from a forbidden region.

Definition 2-7: Vertex and Extent of a Nearness Region

The *jth* nearness region, $R^j(d^j,V^j,E^j)$, formed from forbidden region $R'(\sigma,V',E')$ with vertex $V'(v_1', \ldots, v_N')$ and extent $E'(e_1', \ldots, e_N')$, is a region with

$$d^j = j$$

vertex $V^j(v_1^j, \ldots, v_N^j)$ such that

$$v_j^j = v_j' - 1$$

$$\forall_{k \neq j \in \{1 \ldots N\}} : v_k^j = v_k'$$

and extent $E^j(e_1^j, \ldots, e_N^j)$ such that

$$e_j^j = v_j'$$

$$\forall_{k \neq j \in \{1 \ldots N\}} : e_k^j = e_k'.$$

□

Nearness regions of degree ζ are the intersection of ζ distinct nearness regions. Points within nearness regions of degree ζ model concurrency states where ζ processes are blocked by the vertex hyperplanes of ζ distinct forbidden regions. Nearness regions generated directly from forbidden regions are called nearness regions of degree 1. Associated with a nearness region of degree ζ are two disjoint sets called the *blocked set* B and the *invariant set* I such that $i \in B$ if and only if process i is blocked within the region, and $i \in I$ if and only if extent coordinate $e_i = \infty$. Informally, $i \in I$ if and only if process i is not blocked in the nearness region and process i never releases a semaphore on which a process $j \in B$ is blocked.

We can also associate blocked and invariant sets with a point P contained in a nearness region. These are denoted $B(P)$ and $I(P)$, respectively.

A deadlock region is a nearness region of degree ζ that models states in which one or more processes are blocked forever.

Definition 2-8: Deadlock Region

A deadlock region is a tuple (I,B,V,E), where I is the invariant set, B is the blocked set, V is the vertex, E is the extent, such that

$$B \cup I = \{1, \ldots, N\}.$$

□

The final region of interest is the unsafe region. Unsafe regions model concurrency states from which deadlock is inevitable.

Definition 2-9: Unsafe Region

An unsafe region is a triple (D,V,E), where D is a deadlock region, V is the vertex, and E is the extent.

□

Definition 2-10: Vertex and Extent of an Unsafe Region

Let the nearness regions of degree 1 whose intersection is the deadlock region D be $R^j(d^j, V^j, E^j)$. Let $\max(\varnothing) = 0$. Then the vertex of the unsafe region U formed from deadlock region D is a point $V^U(v_1^U, \ldots, v_N^U)$ such that

$$\forall_{i \in \{1 \ldots N\}} : v_i^U = \max(v_i^k, k \in B^D \wedge k \neq i).$$

The extent of U is the extent of D.

□

2.3.2. Constructing the Model

Consider the problem of building a geometric model of the example program in Figure 2-3. The progress graph of the program extends to infinity along each coordinate axis, and thus defines an infinite space containing an infinite number of forbidden regions. Our first task is to determine some finite subset of the infinite set of forbidden regions that is sufficiently large to tell us whether the program deadlocks. This problem is called the *finite models problem*. For general semaphore programs the finite models problem is open. We will discuss the finite models problem in greater detail in Chapter 4.

For the example program, inspection of the progress graph shows that the forbidden regions whose vertices lie in the space defined by unfolding the first cycle of each process are a sufficiently large collection of forbidden regions. Thus the geometric model of the example program consists of three forbidden regions.

Forbidden Regions			
Number	Semaphore	Vertex	Extent
R_1	a	(0,2)	(1,∞)
R_2	c	(0,3)	(4,∞)
R_3	d	(3,0)	(∞,4)

Our next task is to replace each of the infinite extent coordinates with a finite value that is large enough to render correct conclusions during the analysis of the model. Carson solves this problem in [CARS84] with his notion of a point called "effective" infinity, denoted $\infty(\infty_i, \ldots, \infty_N)$. Carson has developed a rule for defining effective infinity for general semaphore programs. If the number of

statements in process i is k_i, and the model was built by unfolding process i c_i times, then effective infinity is defined by

$$\forall_{i \in \{1...N\}} : \infty_i = k_i(c_i + 1) + 1.$$

For the example program, each process contains four statements, and each process is unfolded 1 time. Thus $\infty = (9,9)$. Given a location for effective infinity, we complete the construction of the geometric model by replacing each *ith* infinite extent coordinate with ∞_i.

Forbidden Regions $\infty = (9,9)$			
Number	Semaphore	Vertex	Extent
R_1	a	(0,2)	(1,9)
R_2	c	(0,3)	(4,9)
R_3	d	(3,0)	(9,4)

To bolster the reader's intuition, we show in Figure 2-7, a part of the progress graph for the program in Figure 2-3. R_i denotes the *ith* forbidden region. The points corresponding to reachable states are represented as solid dots and labeled with the associated states.

2.3.3. Modeling Power and Complexity

Although the theory of geometric models has been developed specifically for semaphore programs, Carson argues in [CARS84] that the theory could be extended to the the class of concurrent programs that synchronize with conditional critical regions.

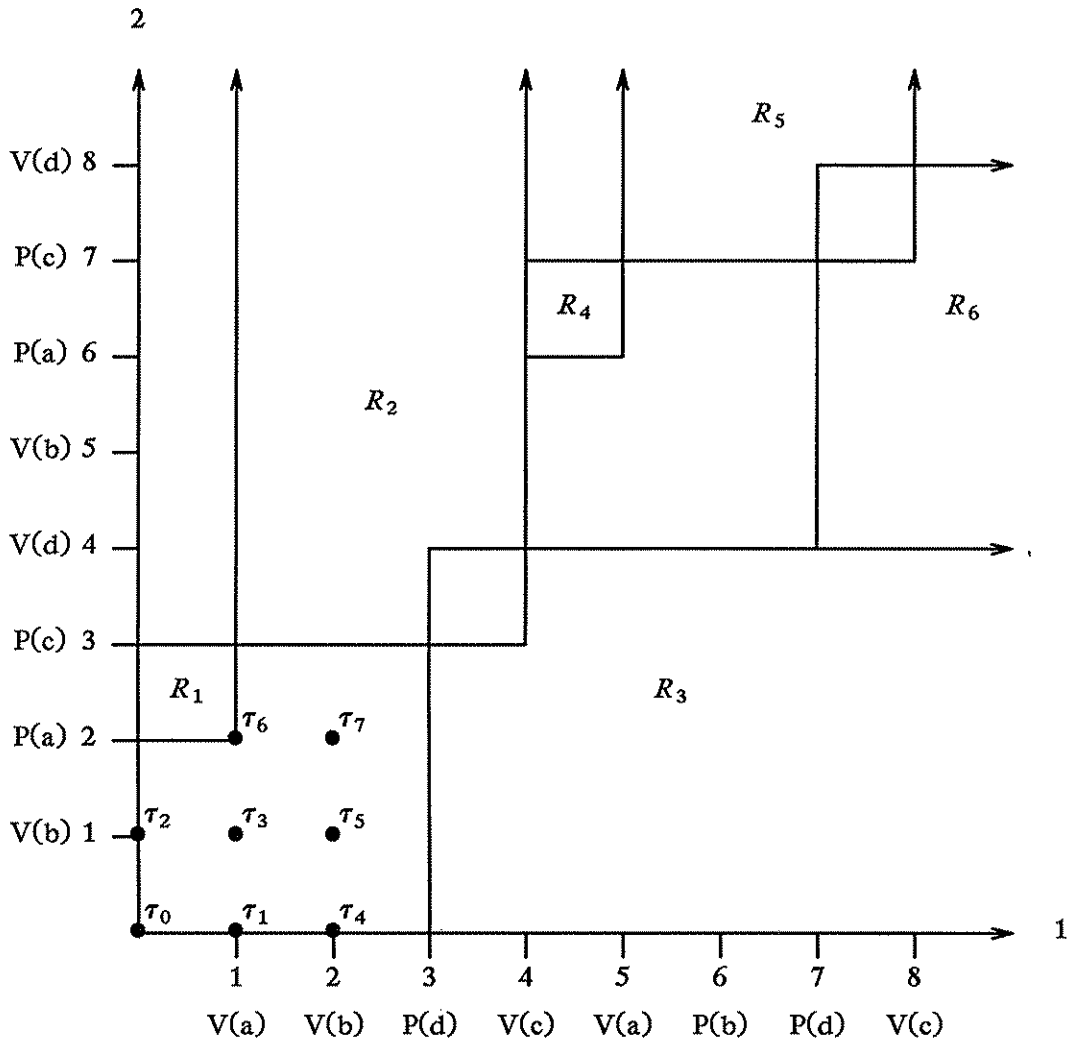


Figure 2-7: *A Progress Graph.*

Carson also shows that constructing a geometric model from the text of a semaphore program requires worst-case time exponential in the number of processes.

2.3.4. Analyzing the Model

To analyze a geometric model for deadlock, we generate all nearness regions of degree 1 to N, and then check the union of the invariant set and the blocked set for each region. If the union is the the set of all processes, then the region is a deadlock region.

Consider how we would analyze the geometric model of our running example for deadlock. Using the definitions of the vertex and extent of a nearness regions, we see that there are three nearness regions of degree 1.

Nearness Regions of Degree 1 $\infty = (9,9)$					
Number	Forbidden Region	Vertex	Extent	Blocked Set	Invariant Set
N_1	R_1	(0,1)	(1,2)	{2}	∅
N_2	R_3	(0,2)	(4,3)	{2}	∅
N_3	R_4	(2,0)	(3,4)	{1}	∅

Since the invariant sets of all nearness regions of degree 1 are empty, none of the regions are deadlock regions. Now, taking the intersections of all nearness regions of degree 1, we see that there is one nearness region of degree 2, D_1, formed by the intersection of nearness regions N_2 and N_3.

Nearness Regions of Degree 2 $\infty = (9,9)$					
Deadlock Region	Parents (deg 1)	Vertex	Extent	Blocked Set	Invariant Set
D_1	N_2,N_3	(2,2)	(3,3)	{1,2}	$\varnothing$

Since the union of the invariant set and the blocked set associated with region D_1 is the set of all processes, we conclude that D_1 is a deadlock region. Since the blocked set is the set of all processes, we conclude that D_1 is a total deadlock region. Inspection shows that the vertex of D_1 is reachable. We note in passing that the definition of unsafe regions can be applied to show that the region with vertex (0,0) and extent (3,3) is an unsafe region. In other words, the program inevitably deadlocks from the initial state.

To bolster the intuition of the reader, we have annotated the progress graph of the example program with the associated nearness regions. N_i in Figure 2-8 denotes the nearness region of degree 1 associated with forbidden region R_i and D_i denotes a total deadlock region.

2.3.5. Decision Power and Complexity

Reachability, total deadlock, and partial deadlock are decidable from the geometric model of a semaphore program in worst-case time exponential in the number of processes[CARS84]. Reachability of a point P is determined by taking an inverse transformation of the model, and then generating the unsafe regions associated with the transformed model. If P is contained in an unsafe region of the transformed model, then P is unreachable from the origin of the original

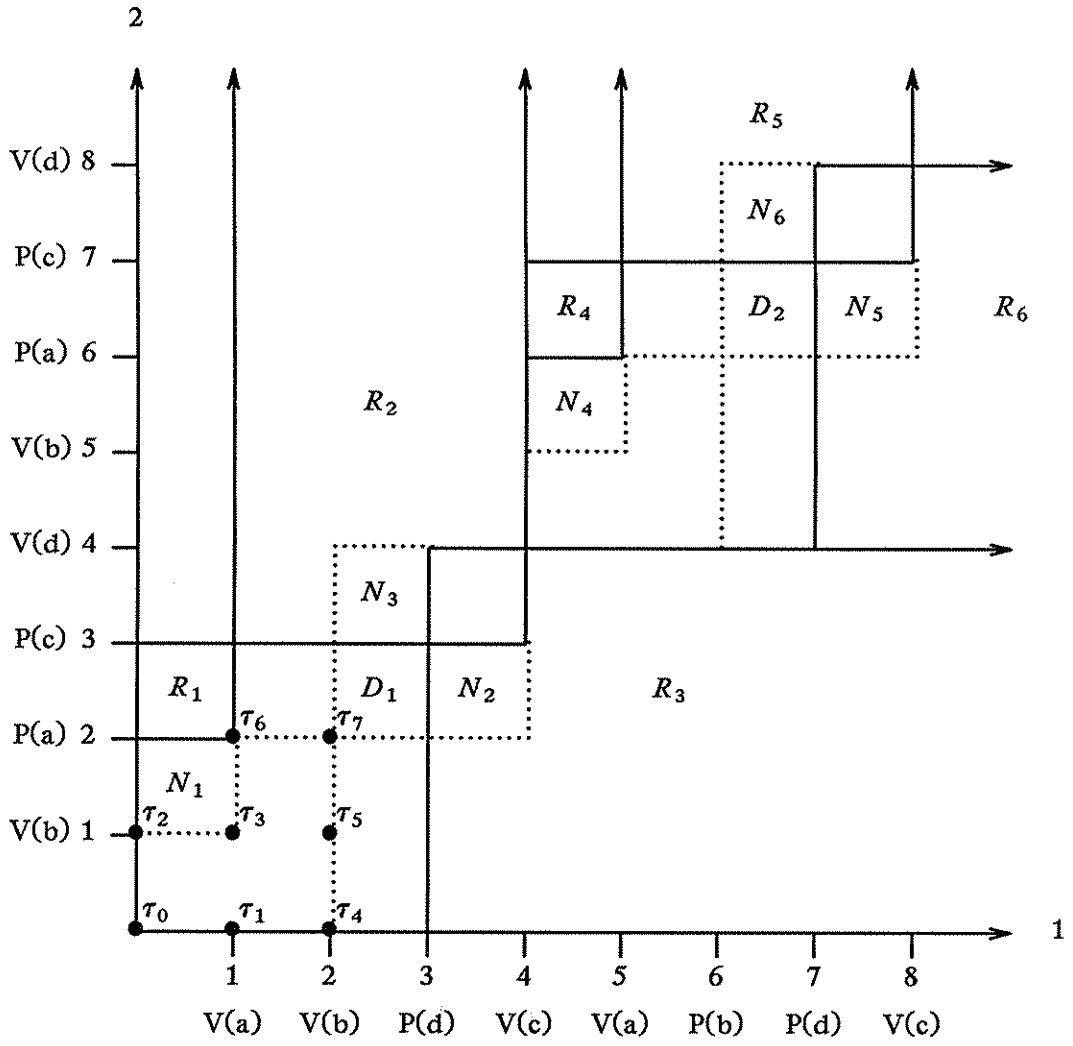


Figure 2-8: *A Progress Graph Annotated with Nearness Regions.*

model. Thus reachability in the geometric model reduces to unsafeness, a most interesting result. The reader is referred to [CARS84] for details.

2.4. Petri Net Models

A Petri net is a static graph structure with well-defined dynamic properties. Like semaphore programs, Petri nets define a class of concurrent systems. In fact, the class of systems defined by Petri nets properly contains the class of systems defined by semaphore programs[PETE81]. For this reason, we can use Petri nets to model and analyze semaphore programs.

Modern interest in Petri nets dates from the writings of Karp and Miller[KARP69], Holt and Commoner[HOLT70], and Commoner and Holt[COMM72], where it was first shown that Petri nets admit interesting theory as well as useful applications.

Since much of the theory has been developed by mathematicians and others outside the mainstream of computer science, Petri nets come with their own terminology and notation different from (and in some cases conflicting with) that in the traditional concurrent programming and operating systems literature. In this section, we give the basic terminology and notation from Petri net theory, and we discuss how Petri nets can be used to model and analyze semaphore programs. We direct the reader interested in learning more about Petri nets to Peterson's excellent introductory text[PETE81].

2.4.1. Definitions

Using notation developed in [ARAK77], [JANT79], and [YAMA84], let W denote the set of nonnegative integers, and let W^n denote the set of n-dimensional W-valued vectors.

A Petri net $N = (\Pi,\Sigma,F,B)$ is a 4-tuple where $\Pi = \{p_1, \ldots, p_{|\Pi|}\}$ is a finite set of *places*, $\Sigma = \{t_1, \ldots, t_{|\Sigma|}\}$ is a finite set of *transitions*, $F : \Sigma \rightarrow W^{|\Pi|}$ is a *forwards incidence function*, and $B : \Sigma \rightarrow W^{|\Pi|}$ is a *backwards incidence function*.

The *ith* coordinates of $F(t)$ and $B(t)$ are denoted $F(p_i,t)$ and $B(t,p_i)$ respectively. In this work we consider only those nets where $F(p_i,t)$ and $B(t,p_i)$ are either zero or unity. If $F(p,t) = 1$ then p is an *input* place of transition t, and t is an *output* transition of place p. If $B(t,p) = 1$ then p is an output place of transition t, and t is an input transition of place p. Let A be a nonempty set of places (transitions). Then $\bullet A$ denotes the set of input transitions (places) of the places (transitions) in A and $A\bullet$ denotes the set of output transitions (places) of the places (transitions) in A. Similarly, if p (t) is a place (transition), we let $\bullet p$ ($\bullet t$) denote the set of input transitions (places) and we let $p\bullet$ ($t\bullet$) denote the set of output transitions (places).

A *marking* of net N, $M_N : \Pi \rightarrow W$, is represented by a $|\Pi|$-dimensional vector, with $M(p)$ denoting the *token count* of place p in marking M. A transition t is *enabled* in marking M if $M \geqslant F(t)$. The *firing* of an enabled transition t in marking M leads to a new marking $M' = M + \bar{t}$, where $\bar{t} = B(t) - F(t)$. For a place p, $M'(p) = M(p) + \bar{t}(p)$, where $\bar{t}(p) = B(t,p) - F(p,t)$. Let $M \xrightarrow{t}$ denote that transition t is enabled in marking M and let $M \xrightarrow{t} M'$ denote that the firing of enabled transition t in marking M leads to marking M'.

Less formally, a transition t is enabled in a marking M if every input place of t contains at least one token. Notice that a transition with no input places is enabled in every marking. When we fire t in marking M, we remove a token from every input place of t and add a token to every output place of t to get the new marking M'.

The notion of firing can be extended to finite sequences of transitions. The empty sequence λ is enabled in every marking M, and $\overline{\lambda} = 0$. For a sequence of transitions $w \in \Sigma^*$ and a transition t, $\overline{tw} = \bar{t} + \overline{w}$, and the sequence tw is enabled in M if t is enabled in marking M and w is enabled in marking $(M + \bar{t})$. The firing of an enabled sequence w from a marking M leads to a new marking $M' = M + \overline{w}$. For a place p, $M'(p) = M(p) + \overline{w}(p)$. Let $M \xrightarrow{w}$ denote that sequence w is enabled in marking M and let $M \xrightarrow{w} M'$ denote that the firing of enabled sequence w in marking M leads to marking M'.

A marking M' is *reachable* from marking M if there exists an enabled sequence w such that $M \xrightarrow{w} M'$. If M' is reachable from M, we sometimes write $M \xrightarrow{*} M'$. The *reachability set* for a Petri net $N = (\Pi, \Sigma, F, B)$ and a marking M consists of all markings reachable from M, is denoted by $R(N,M)$, and is defined by

$$R(N,M) = \{M' \mid M \xrightarrow{*} M'\}.$$

The *Parikh* mapping $\Psi : \Sigma^* \rightarrow W^{|\Sigma|}$ is defined such that $\Psi(w)(t)$ is the number of occurrences of transition t in sequence w. For a set of transitions T, $\Psi(w)(T)$ is defined by

$$\Psi(w)(T) = \sum_{t \in T} \Psi(w)(t).$$

Notice that for a place p and a sequence w

$$\overline{w}(p) = \Psi(w)(\bullet p) - \Psi(w)(p\bullet).$$

For sequences w and w', if $\Psi(w) = \Psi(w')$, then w' is a *rearrangement* of w.

A transition t is *dead* in a marking M if there exists no sequence w such that $M \xrightarrow{w} M'$ and t is enabled in M'. A transition t is *live* in a marking M if for all M' reachable from M, there exists a sequence w such that $M' \xrightarrow{w} M''$ and t is enabled in M''. A marking M is dead if every transition is dead in M. A marking M is live if every transition is live in M.

Petri nets are represented by directed bipartite graphs where a *node* in the graph is either a place or a transition. Places are represented by circles, transitions by bars, and tokens by solid dots inside the places. A *path* c in a Petri net $N = (\Pi,\Sigma,F,B)$ is a sequence of nodes $c_1c_2\cdots c_k$ such that $c_i \in \bullet c_{i+1}$ for $1 \leqslant i < k$. If $c_1 = c_k$ then c is a *circuit*. If in addition $c_i \neq c_j$ for $1 \leqslant i < j < k$, then c is a *simple* circuit. If the set of places in c does not properly include the set of places in any other circuit, then c is a *minimal* circuit. For some c_i in a circuit c, c_{i-1} is the *immediate predecessor* in c for $1 < i \leqslant k$ and c_{i+1} is the *immediate successor* in c for $1 \leqslant i < k$.

If P is the set of places in a path c, then the token count of c in marking M is denoted $M(c)$ and is defined by

$$M(c) = \sum_{p \in P} M(p).$$

If the token count of a path (circuit) in marking M is greater than zero, then the path (circuit) is *marked* in M.

2.4.2. Constructing the Model

The construction of Petri net models of semaphore programs is straightforward and well-known[KELL76, LAUT74, PETE81]. Consider the problem of constructing a Petri net model of the program in Figure 2-3.

First we model each process, ignoring the synchronization constraints imposed by the semaphore operations. We associate a place/transition pair with each statement of the program, as shown in Figure 2-9(a). Places $p_1 - p_4$ track the value of the program counter in process 1, while places $p_5 - p_8$ track the value of the program counter in process 2. Transitions $t_1 - t_4$ represent the P and V statements in process 1, while transitions $t_5 - t_8$ represent the P and V statements in process 2. The sequential nature of each process is modeled with the directed arcs. The tokens in places p_1 and p_5 model the initial value of the program counter in processes 1 and 2, respectively.

Next, we add a place for each semaphore and we add directed arcs to model the synchronization constraints imposed by the various semaphore operations. This is shown in Figure 2-9(b). Places $p_9 - p_{12}$ represent semaphores $a - d$ respectively. Each of these is marked with the number of tokens equal to the initial value of its respective semaphore (zero in this case). If transition t models a P operation on a semaphore modeled by place p, then we add the directed arc (p,t) to the Petri net. If transition t models a V operation on a semaphore modeled by place p, then we add the directed arc (t,p) to the Petri net. The resulting Petri net N and its *initial marking* M_0 is called the Petri net model of the program. It is not hard to see how this construction generalizes to any semaphore program.

Legend					
Process 1		Process 2		Semaphores	
transition	statement	transition	statement	place	semaphore
t_1	V(a)	t_5	V(b)	p_9	a
t_2	V(b)	t_6	P(a)	p_{10}	b
t_3	P(d)	t_7	P(c)	p_{11}	c
t_4	V(c)	t_8	V(d)	p_{12}	d

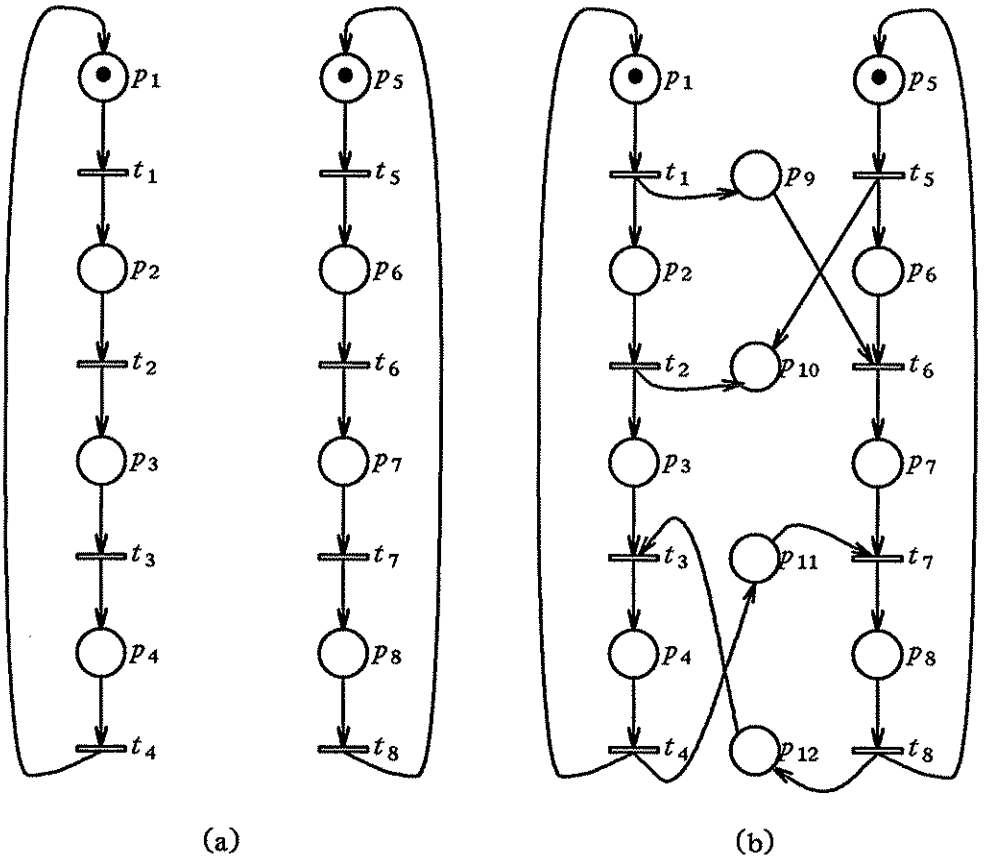


Figure 2-9: *A Petri Net Model.*

The mapping between a semaphore program C with initial state τ_0 and its Petri net model N with initial marking M_0 is straightforward. A state τ of C corresponds to some marking M of N; the initial state τ_0 of C corresponds to the initial marking M_0 of N. A statement S in C corresponds to some transition t in N.

Statement S is dead in state τ if and only transition t is dead in marking M; statement S is live in state τ if and only if transition t is live in marking M. A sequence of statements $w = S_1 S_2 \cdots S_k$ is executable in state τ if and only transition sequence $v = t_1 t_2 \cdots t_k$ is enabled in marking M and for all i, $S_i \in w$ corresponds to $t_i \in v$.

State τ' is reachable from state τ if and only if marking M' is reachable from marking M. State τ is a total deadlock state if and only if marking M is dead. If state τ is a partial deadlock state then marking M is not live; the converse is not necessarily true. Finally program C is deadlock-free if and only if initial marking M_0 of N is live.

2.4.3. Modeling Power and Complexity

Petri nets are capable of modeling the class of the systems called *additive monotone* systems by Keller in [KELL77]. The class of additive monotone systems properly includes the class of semaphore systems. That is, there are systems that can be modeled with Petri nets that cannot be modeled with semaphore programs[PETE81].

It should be clear from our discussion that constructing Petri net models of semaphore programs requires time and space polynomial in the number of processes.

2.4.4. Analyzing the Model

There are two basic approaches to analyzing a Petri net N and a marking M for liveness or reachability. With the first approach we analyze the directed graph representation of the Petri net, and from the structural properties of the graph we make inferences about the dynamic behavior of the Petri net. For example, in Chapter 6 we develop necessary and sufficient conditions for the liveness of markings of a subclass of Petri nets. These conditions are dependent only on the static properties of the net N and its marking M.

With the second approach, we generate a finite representation of the possibly infinite reachability set, from which we make inferences about the dynamic behavior of the net. If the reachability set of a Petri net N and marking M is finite, then the reachability tree of Karp and Miller[KARP69] can be used to decide reachability and liveness[PETE81]. In general though, the reachability tree cannot be used to decide liveness or reachability.

Since the program in Figure 2-3 has a finite number of reachable states, the reachability set of its Petri net model is finite, and we can use the reachability tree to decide liveness. The reachability tree for the Petri net model in Figure 2-9 is shown in Figure 2-10. A node in the tree consists of a label and marking, or simply a label enclosed in parentheses if the node is a duplicate of another node in the tree. Marking M_i corresponds to state τ_i of the example program. The son of a father represents the marking reachable by firing a transition that is enabled in the marking represented by the father. The arc between father and son is labeled with the enabled transition. A detailed and very readable description of the algorithm for constructing the reachability tree is given in [PETE81].

Legend					
Process 1		Process 2		Semaphores	
transition	statement	transition	statement	place	semaphore
t_1	V(a)	t_5	V(b)	p_9	a
t_2	V(b)	t_6	P(a)	p_{10}	b
t_3	P(d)	t_7	P(c)	p_{11}	c
t_4	V(c)	t_8	V(d)	p_{12}	d

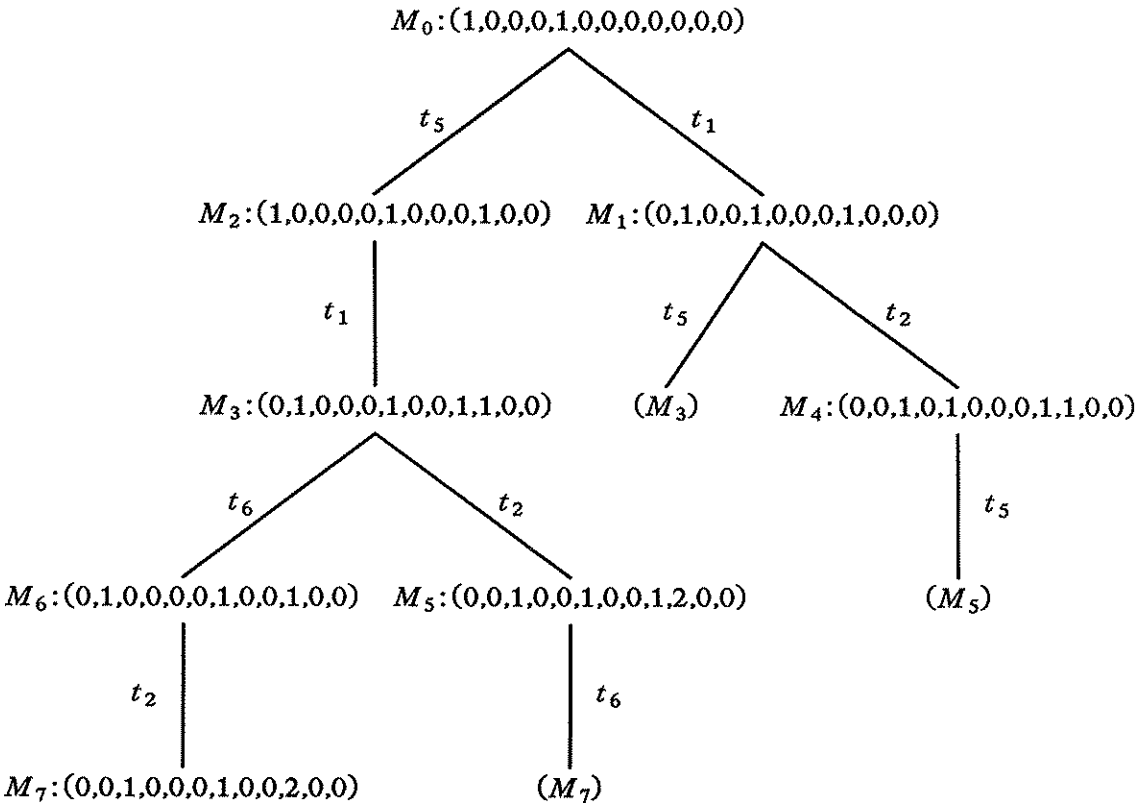


Figure 2-10: *A Reachability Tree.*

Inspection of the reachability tree shows that the reachable marking M_7 is dead, since no transitions are enabled in M_7. Partial deadlock states can also be detected by a traversal of the tree.

2.4.5. Decision Power and Complexity

Liveness of markings of Petri net models of semaphore programs is known to be decidable[HERZ77, HERZ79]; no complexity results for the analysis are reported. Reachability is known to be decidable[KOSA82, MAYR81, MAYR84]; the complexity of the analysis is unknown, although it is known to require at least exponential space[LIPT76].

2.5. Chapter Summary

We have discussed concepts and notations for semaphore programs and algebraic, geometric, and Petri net models in preparation for the presentation of the results that follow in the remaining chapters.

We have seen that algebraic models of semaphore programs are simple to construct; the price we pay for this simplicity is an inability to infer either reachability or freedom from partial deadlock from the model. Geometric models have greater decision power than algebraic models; the price we pay is the worst-case exponential complexity of building the model. Petri net models are simple to construct and have the same decision power as the geometric model; the worst-case complexity of analysis is at least exponential.

CHAPTER 3

A Generalized Deadlock Predicate

As we noted in Chapter 2, the algebraic model is incomplete for two reasons. First, because of the possibility of feasible yet unreachable total deadlock states, satisfiability of the deadlock predicate is not a sufficient condition for total deadlock[CLAR80]. Second, the total deadlock predicate cannot be used to detect partial deadlock[CARS84].

Clarke has addressed the first problem by developing an iterative procedure to synthesize from the program text a strongest resource invariant that is satisfied only by reachable states. With this strongest resource invariant, satisfiability of the total deadlock predicate becomes a necessary and sufficient condition for total deadlock.

We address the second problem by developing a technique for synthesizing from the text of a semaphore program a predicate, the satisfiability of which is a necessary and sufficient condition for the existence of reachable partial or total deadlock states. We call this predicate the *generalized deadlock predicate* [O'HA86]. Our method is a variant of Clarke's technique for generating strongest resource invariants. In essence, our method uses predicate transformers to derive a predicate that is satisfied by the states where one or more processes are deadlocked. Other methods for treating freedom from total and partial deadlock using predicate transformers have been described in [FLON81] and [LAMS79]. These methods, unlike our method, use predicate transformers to derive a predicate that is satisfied by the

set of *initial* states from which a process is guaranteed not to deadlock.

3.1. Clarke's Strongest Total Deadlock Predicate

As in [CLAR80] and [DIJK76], we associate a predicate J with the set of states that satisfy J. Let T be the set of all program states. Then 2^{T} corresponds to the set of all possible predicates, **false** corresponds to the empty set of states, and **true** corresponds to the universal set of states, T. Also, we can interpret logical operations on predicates in terms of set-theoretic operations on subsets of T. Thus, logical disjunction becomes set-theoretic union, logical conjunction becomes set-theoretic intersection, and logical negation becomes set-theoretic complementation.

Let F be a functional, $F : 2^{\mathrm{T}} \rightarrow 2^{\mathrm{T}}$, that maps predicates into predicates. If J is a predicate and $F(J) = J$, then J is said to be a *fixpoint* for the functional F. Clarke's technique for constructing strongest resource invariants is based on a view of a strongest resource invariant as a fixpoint of a functional F.

$SP[A](J)$ denotes the *strongest postcondition* [CLAR80] corresponding to the statement A and the *precondition* J. If we associate with predicate J the set of states that satisfy it, then $SP[A](J)$ is defined by

$$SP[A](J) = \{(pc_1, \ldots, pc_N : A(s)) \mid (pc_1, \ldots, pc_N : s) \in J\}.$$

Given this notion of the strongest postcondition, the *fixpoint functional* $F : 2^{\mathrm{T}} \rightarrow 2^{\mathrm{T}}$ is defined by

$$F(J) = J_0 \vee J \vee \left\{ \bigvee_{i=1}^{N} \left\{ \bigvee_{k=1}^{k_i} SP[S_i^k](pre(S_i^k) \wedge \neg b_i^k \wedge J) \right\} \right\}.$$

where J is a predicate and J_0 is a predicate describing the initial state of the program. Intuitively we think of $F(J)$ as the union of the initial state, the set of states that satisfy J, and the reachable immediate successors of the states that satisfy J.

The strongest resource invariant $F^*(\mathbf{false})$ is defined by

$$F^*(\mathbf{false}) = \bigcup_{i=0}^{\infty} F^i(\mathbf{false}),$$

where $F^0(J) = J$ and $F^{i+1}(J) = F(F^i(J))$. Intuitively, we think of the strongest resource invariant as being satisfied by the set of all states reachable from the initial state.

Clarke has noted that $F^*(\mathbf{false})$ converges in a finite number of steps only when the program has a finite number of reachable states.

Given the strongest resource invariant, $F^*(\mathbf{false})$, Clarke constructs a strongest total deadlock predicate

$$D = \bigwedge_{i=1}^{N} \left\{ \bigvee_{j=1}^{k_i} pre(S_i^j) \wedge b_i^j \right\} \wedge F^*(\mathbf{false})$$

which is satisfied by those reachable states where all processes are deadlocked. Its satisfiability is a necessary and sufficient condition for total deadlock. In the next section, we construct a strongest total deadlock predicate for the example program in Figure 2-3.

3.2. Constructing the Strongest Total Deadlock Predicate

Consider the problem of constructing a strongest total deadlock predicate for the example program in Figure 2-3. Recall the example.

```
var a,b,c,d = 0: semaphore

cobegin
        1: cycle
                V(a)        — send message a
                V(b)        — send message b
                P(d)        — wait for message d
                V(c)        — wait for message c
        endcycle
        //
        2: cycle
                V(b)        — send message b
                P(a)        — wait for message a
                P(c)        — wait for message c
                V(d)        — send message d
        endcycle
coend
```

The reachable states of the example program are

$\tau_0 = (1,1;a=0,b=0,c=0,d=0)$ $\tau_4 = (3,1;a=1,b=1,c=0,d=0)$

$\tau_1 = (2,1;a=1,b=0,c=0,d=0)$ $\tau_5 = (3,2;a=1,b=2,c=0,d=0)$

$\tau_2 = (1,2;a=0,b=1,c=0,d=0)$ $\tau_6 = (2,3;a=0,b=1,c=0,d=0)$

$\tau_3 = (2,2;a=1,b=1,c=0,d=0)$ $\tau_7 = (3,3;a=0,b=2,c=0,d=0)$

The progress graph for the example program is shown in Figure 3-1. The reachable states are marked with solid dots and labeled with the corresponding states. Although we will refer to the progress graph often during our discussion, the reader should be aware that the progress graph is not essential to the construction of the strongest total deadlock predicate; we use the progress graph as an exposi-

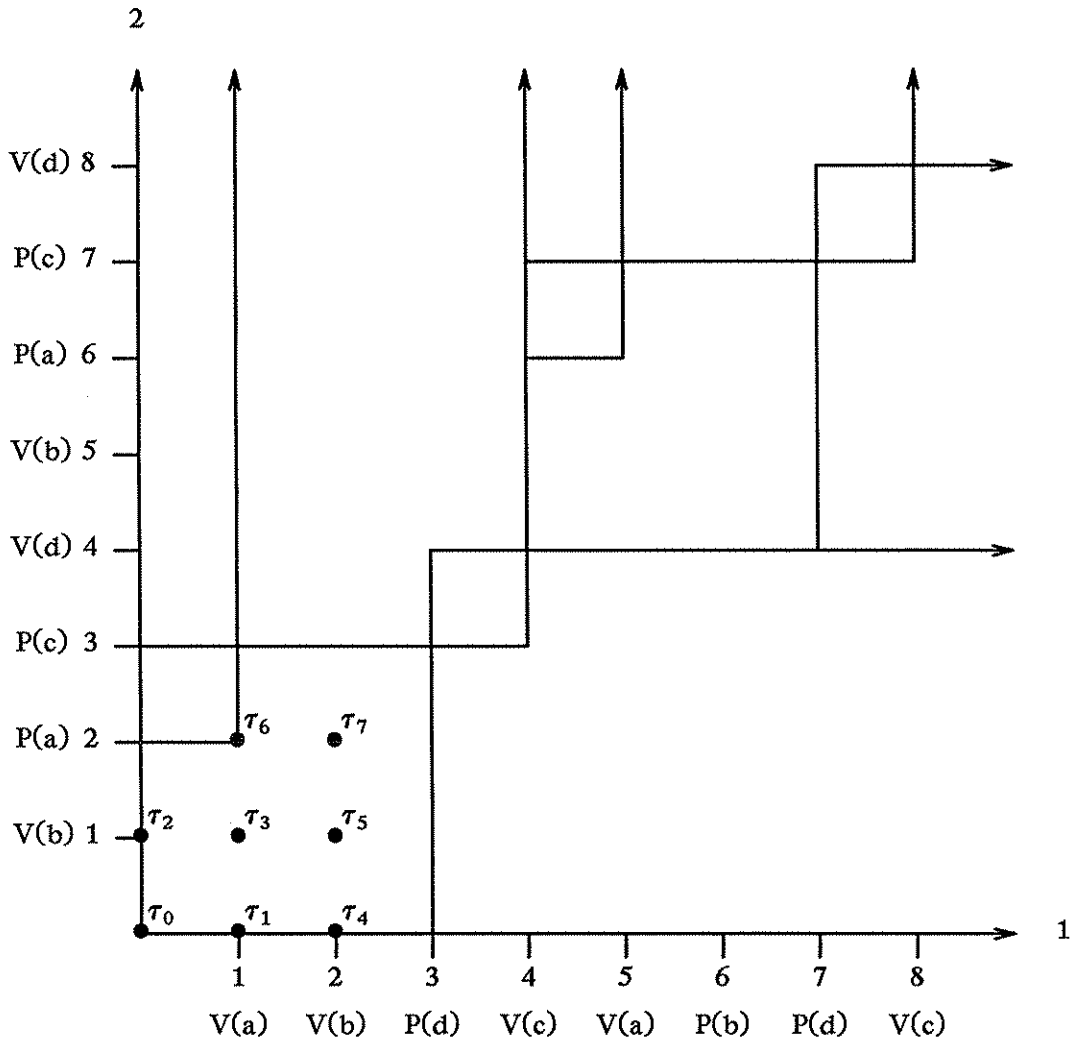


Figure 3-1: *Progress Graph for the Example Program.*

tory device.

Our first task is to construct the fixpoint functional $F(J)$ using the definition in the previous section,

$$
\begin{aligned}
F(J) = J_0 \vee J \\
&\vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge J) \\
&\vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge J) \\
&\vee SP[S_1^3]((a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V) \wedge (d > 0) \wedge J) \\
&\vee SP[S_1^4]((a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V) \wedge J) \\
&\vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge J) \\
&\vee SP[S_2^2]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge (a > 0) \wedge J) \\
&\vee SP[S_2^3]((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge (c > 0) \wedge J) \\
&\vee SP[S_2^4]((b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V) \wedge J)
\end{aligned}
$$

where J_0 is the predicate describing the initial state

$$J_0 = (a_1^V = b_1^V = d_1^P = c_1^V = 0) \wedge (b_2^V = a_2^P = c_2^P = d_2^V = 0).$$

Next, we construct the strongest resource invariant $F^*(\mathbf{false})$ by applying the fixpoint functional, starting with $F^0(\mathbf{false})$, until $F^i(\mathbf{false}) = F^{i+1}(\mathbf{false})$. By definition,

$$F^0(\mathbf{false}) = \mathbf{false}$$

which is satisfied by no states. Applying the fixpoint functional again, we get

$$F^1(\mathbf{false}) = J_0$$

which is satisfied by τ_0. Applying the fixpoint functional again gives

$$F^2(\mathbf{false}) = F^1(\mathbf{false})$$

$$\vee\ SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_1^3]((a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V) \wedge (d > 0) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_1^4]((a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_2^2]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge (a > 0) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_2^3]((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge (c > 0) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_2^4]((b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V) \wedge F^1(\textbf{false}))$$

which reduces to

$$F^2(\textbf{false}) = F^1(\textbf{false})$$
$$\vee\ SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_1^2](\textbf{false})$$
$$\vee\ SP[S_1^3](\textbf{false})$$
$$\vee\ SP[S_1^4](\textbf{false})$$
$$\vee\ SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^1(\textbf{false}))$$
$$\vee\ SP[S_2^2](\textbf{false})$$
$$\vee\ SP[S_2^3](\textbf{false})$$
$$\vee\ SP[S_2^4](\textbf{false})$$

which further reduces to

$$F^2(\textbf{false}) = F^1(\textbf{false})$$
$$\vee\ SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^1(\textbf{false})) \quad (3\text{-}1)$$
$$\vee\ SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^1(\textbf{false})) \quad (3\text{-}2)$$

Strongest postcondition 3-1 is satisfied by τ_1, and strongest postcondition 3-2 is satisfied τ_2. Thus, $F^2(\textbf{false})$ is satisfied by τ_0, τ_1, and τ_2.

Applying the fixpoint functional again we have

$$
\begin{aligned}
F^3(\mathbf{false}) = J_0 & \\
& \vee F^2(\mathbf{false}) \\
& \vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_1^3]((a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V) \wedge (d > 0) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_1^4]((a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_2^2]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge (a > 0) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_2^3]((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge (c > 0) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_2^4]((b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V) \wedge F^2(\mathbf{false}))
\end{aligned}
$$

which reduces to

$$
\begin{aligned}
F^3(\mathbf{false}) = J_0 & \\
& \vee F^2(\mathbf{false}) \\
& \vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_1^3](\mathbf{false}) \\
& \vee SP[S_1^4](\mathbf{false}) \\
& \vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^2(\mathbf{false})) \\
& \vee SP[S_2^2](\mathbf{false}) \\
& \vee SP[S_2^3](\mathbf{false}) \\
& \vee SP[S_2^4](\mathbf{false})
\end{aligned}
$$

which further reduces to

$$F^3(\mathbf{false}) = J_0$$

$$\vee F^2(\mathbf{false})$$

$$\vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \quad (3\text{-}3)$$

$$\vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^2(\mathbf{false})) \quad (3\text{-}4)$$

$$\vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^2(\mathbf{false})) \quad (3\text{-}5)$$

Strongest postcondition 3-3 is satisfied by τ_1 and τ_3. Strongest postcondition 3-4 is satisfied by τ_4. Strongest postcondition 3-5 is satisfied by τ_2 and τ_3. Thus, $F^3(\mathbf{false})$ is satisfied by τ_0, τ_1, τ_2, τ_3, and τ_4.

Following the same procedure for $F^4(\mathbf{false})$, we see that

$$F^4(\mathbf{false}) = J_0 \vee F^3(\mathbf{false})$$

$$\vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^3(\mathbf{false})) \quad (3\text{-}6)$$

$$\vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^3(\mathbf{false})) \quad (3\text{-}7)$$

$$\vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^3(\mathbf{false})) \quad (3\text{-}8)$$

$$\vee SP[S_2^2]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge (a > 0) \wedge F^3(\mathbf{false})) \quad (3\text{-}9)$$

Strongest postcondition 3-6 is satisfied by τ_1 and τ_3. Strongest postcondition 3-7 is satisfied by τ_4 and τ_5. Strongest postcondition 3-8 is satisfied by τ_2, τ_3, and τ_5. Strongest postcondition 3-9 is satisfied by τ_6. Thus, $F^4(\mathbf{false})$ is satisfied by τ_0, τ_1, τ_2, τ_3, τ_4, τ_5, and τ_6.

Following the same procedure for $F^5(\mathbf{false})$, we see that

$$F^5(\mathbf{false}) = J_0 \vee F^4(\mathbf{false})$$

$$\vee SP[S_1^1]((a_1^V = b_1^V = d_1^P = c_1^V) \wedge F^4(\mathbf{false})) \quad (3\text{-}10)$$

$$\vee SP[S_1^2]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge F^4(\mathbf{false})) \quad (3\text{-}11)$$

$$\vee SP[S_2^1]((b_2^V = a_2^P = c_2^P = d_2^V) \wedge F^4(\mathbf{false})) \quad (3\text{-}12)$$

$$\vee SP[S_2^2]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge (a > 0) \wedge F^4(\mathbf{false})) \quad (3\text{-}13)$$

Strongest postcondition 3-10 is satisfied by τ_1 and τ_3. Strongest postcondition 3-11 is satisfied by τ_4, τ_5, and τ_7. Strongest postcondition 3-12 is satisfied by τ_2, τ_3, and τ_5. Strongest postcondition 3-13 is satisfied by τ_6 and τ_7. Thus, $F^5(\mathbf{false})$ is satisfied by τ_0, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6, and τ_7.

Following the same procedure for $F^6(\mathbf{false})$, we see that

$$F^6(\mathbf{false}) = F^5(\mathbf{false})$$

and we are done. The strongest resource invariant $F^*(\mathbf{false}) = F^5(\mathbf{false})$ is satisfied by the states τ_0, τ_1, τ_2, τ_3, τ_4, τ_5, τ_6, and τ_7, which are the reachable states.

Given the strongest resource invariant $F^*(\mathbf{false})$, constructing the strongest total deadlock predicate is trivial. We merely replace the conjunction of semaphore invariants in the deadlock predicate with $F^*(\mathbf{false})$.

3.3. A Generalized Deadlock Predicate

Clarke's technique addresses one of the problems with the deadlock predicate by strengthening the semaphore invariants to exclude unreachable states so that satisfiability of the strongest total deadlock predicate is a necessary and sufficient condition for total deadlock. However, the strongest total deadlock predicate cannot be used to detect partial deadlock. In this section, we present a technique for constructing a generalized deadlock predicate, the satisfiability of which is a necessary and sufficient condition for total or partial deadlock.

We let $wp[A](J)$ denote the *weakest precondition* [DIJK76] corresponding to the statement A and the *postcondition* J. If we associate with predicate J the set of states that satisfy it, then $wp[A](J)$ is defined by

$$wp[A](J) = \{(pc_1, \ldots, pc_N; s) \mid (pc_1, \ldots, pc_N; A(s)) \in J\}.$$

We start with a weak predicate called the *blocking predicate*. The blocking predicate for process i is defined by

$$B_i = \left(\bigvee_{k=1}^{k_i} pre(S_i^k) \wedge b_i^k \right) \wedge F^*(\mathbf{false})$$

Informally, the first term of the blocking predicate is satisfied by those states where process i is attempting to execute a statement whose blocking condition is true. The conjunction of the strongest resource invariant excludes all of the unreachable states. Thus B_i is satisfied by the set of reachable states where process i is blocked. The blocking predicate for all processes

$$B = \bigvee_{i=1}^{N} B_i ,$$

is satisfied by the set of reachable states where at least one process is blocked. Since blocking is a necessary condition for deadlock, the satisfiability of B is a necessary condition for deadlock. For the satisfiability of B to be a sufficient condition for deadlock, we must strengthen it to include only those reachable states where at least one process is blocked forever. Our strengthening technique uses a variant of Clarke's fixed point functional.

The functional $G : 2^{\mathrm{T}} \rightarrow 2^{\mathrm{T}}$ is defined by

$$G(J) = J \vee \left\{ \bigvee_{i=1}^{N} \left\{ \bigvee_{k=1}^{k_i} wp[S_i^k](post(S_i^k) \wedge J) \wedge F^*(\mathbf{false}) \right\} \right\}.$$

If J is satisfied by a set of reachable states, then intuitively we think of $G(J)$ as the union of the set of states that satisfy J with the set of reachable immediate

predecessors of the states that satisfy J. We state this as a theorem. For a predicate J, we let $IP(J)$ denote the set of states, τ, such that τ is a reachable immediate predecessor of some $\tau' \in J$.

Theorem 3-1: For all $\tau \in \Sigma$, and for all $J \subseteq F^*(\mathbf{false})$, $\tau \in G(J)$ if and only if $\tau \in J$ or $\tau \in IP(J)$.

Proof:

If: Let $\tau = (pc_1, \ldots, pc_N; s)$. If $\tau \in J$, then we are done. If $\tau \in IP(J)$, then let $\tau' = (pc_1, \ldots, pc_N; A(s)) \in J$ be an immediate successor of τ. Clearly $\tau' \in post(A)$ and $\tau' \in J$. Thus

$$\tau' \in post(A) \cap J$$

and by the definition of wp,

$$\tau \in wp[A](post(A) \cap J).$$

Furthermore,

$$\tau \in wp[A](post(A) \cap J) \cap F^*(\mathbf{false})$$

by the assumption that $\tau \in IP(J)$. It follows from the definition of G that $\tau \in G(J)$.

Only If: Let $\tau = (pc_1, \ldots, pc_N; s) \in G(J)$. If $\tau \in J$, then we are done. If $\tau \notin J$, then by the definition of G and wp, there is some reachable $\tau' \in J$ and some statement A such that $\tau' = (pc_1, \ldots, pc_N; A(s))$. If A is a V operation on semaphore σ, then $\sigma(s) \geq 0$ by the reachability of τ, which implies that $\sigma(A(s)) > 0$. If A is a P operation on semaphore σ, then $\sigma(A(s)) \geq 0$ by the reachability of τ', which implies that $\sigma(s) > 0$. In either case, we have a computation $\tau_0 \cdots \tau\tau'$ by the reachability of τ and the definition of a computation. Thus $\tau \in IP(J)$

□

Since the predicate B_i is satisfied by the set of all reachable states where process i is blocked, the predicate $U_i = \neg B_i \wedge F^*(\mathbf{false})$ is satisfied by the set of all reachable states where process i is not blocked. Now, if we iteratively apply the functional G to the predicate U_i, we get

$$G^*(U_i) = \bigcup_{k=0}^{\infty} G^k(U_i),$$

where $G^0(J) = J$, and $G^{k+1}(J) = G(G^k(J))$. Intuitively we think of $G^*(U_i)$ as the union of the set of all reachable states where process i is not blocked with the set of all reachable states where process i is blocked but eventually becomes unblocked. Thus, G^* is a predicate transformer, which when applied to U_i gives a weaker predicate that is satisfied by the reachable states where process i is not deadlocked. We state this as a theorem.

Theorem 3-2: For all $\tau \in \Sigma$, $\tau \in G^*(U_i)$ if and only if τ is reachable and process i is not deadlocked in τ.

Proof:

If: If process i is not deadlocked in τ and τ is reachable, then either $\tau \in U_i$, in which case we are done, or there is some computation $c = \tau \ldots \tau' \tau''$ such that $\tau'' \in U_i$. Now, by Theorem 3-1, the immediate predecessor of τ'' in c, τ', is a member of $G(U_i)$, the immediate predecessor of τ' in c is a member of $G^2(U_i) = G(G(U_i))$, and so on. Eventually, we find some k such that $\tau \in G^k(U_i)$, which implies that $\tau \in G^*(U_i)$.

Only If: If $\tau \in G^*(U_i)$ then there is some $k \geqslant 0$ such that $\tau \in G^k(U_i)$. We use induction on k to show that if $\tau \in G^k(U_i)$, then τ is reachable and process i is not deadlocked in τ.

Basis: If $\tau \in G^0(U_i) = U_i$, then by the definition of U_i, τ is reachable and process i is not deadlocked in τ.

Induction Hypothesis: Suppose that for all $\tau \in \Sigma$, $\tau \in G^k(U_i)$ only if τ is reachable and process i is not deadlocked in τ.

Induction Step: We want to show that for all τ, $\tau \in G^{k+1}(U_i)$ only if τ is reachable and process i is not deadlocked in τ. Now, $G^{k+1}(U_i) = G(G^k(U_i))$. By the induction hypothesis and Theorem 3-1, for all $\tau \in \Sigma$, $\tau \in G(G^k(U_i))$ if and only if $\tau \in G^k(U_i)$ or $\tau \in G^k(IP(U_i))$. If $\tau \in G^k(U_i)$, then we are done. If $\tau \in G^k(IP(U_i))$, then τ is a reachable immediate predecessor of some state in $G^k(U_i)$ where process i is not deadlocked. Thus, τ is reachable and process i is not deadlocked in τ.

□

By Theorem 3-2

$$\tau \in G^*(U_i) \iff (i \text{ not deadlocked in } \tau) \wedge (\tau \text{ reachable})$$

which implies

$$\tau \in \neg G^*(U_i) \iff (i \text{ deadlocked in } \tau) \vee (\tau \text{ not reachable}).$$

By the definition of B_i

$$\tau \in B_i \iff (i \text{ blocked in } \tau) \wedge (\tau \text{ reachable}).$$

Thus

$$\tau \in \neg G^*(U_i) \wedge B_i \iff ((i \text{ deadlocked in } \tau) \vee (\tau \text{ not reachable})) \wedge (i \text{ blocked in } \tau) \wedge (\tau \text{ reachable})$$

which implies

$$\tau \in \neg G^*(U_i) \wedge B_i \iff (i \text{ deadlocked in } \tau) \wedge (i \text{ blocked in } \tau) \wedge (\tau \text{ reachable})$$

which implies

$$\tau \in \neg G^*(U_i) \wedge B_i \iff (i \text{ deadlocked in } \tau) \wedge (\tau \text{ reachable}).$$

Thus, the predicate

$$D_i = \neg G^*(U_i) \wedge B_i$$

is satisfied by the set of reachable states where process i is blocked forever.

Given $G^*(U_i)$ for all processes, we can restate Clarke's strongest total deadlock predicate as

$$D = \bigwedge_{i=1}^{N} D_i,$$

which is satisfied by those reachable states where all processes are deadlocked. Its satisfiability is a necessary and sufficient condition for total deadlock. Furthermore, we can define the generalized deadlock predicate as

$$D^* = \bigvee_{i=1}^{N} D_i,$$

which is satisfied by all reachable states where at least one process is deadlocked. Its satisfiability is a necessary and sufficient condition for partial or total deadlock.

3.4. Constructing the Generalized Deadlock Predicate

Consider the problem of constructing the generalized deadlock predicate for the example program in Figure 2-3. Recall the example.

```
var a,b,c,d = 0: semaphore

cobegin
        1: cycle
                V(a)        — send message a
                V(b)        — send message b
                P(d)        — wait for message d
                V(c)        — wait for message c
        endcycle
        //
        2: cycle
                V(b)        — send message b
                P(a)        — wait for message a
                P(c)        — wait for message c
                V(d)        — send message d
        endcycle
coend
```

The progress graph for the program is shown in Figure 3-2. As before, the reachable points are marked with solid dots and labeled with the corresponding states.

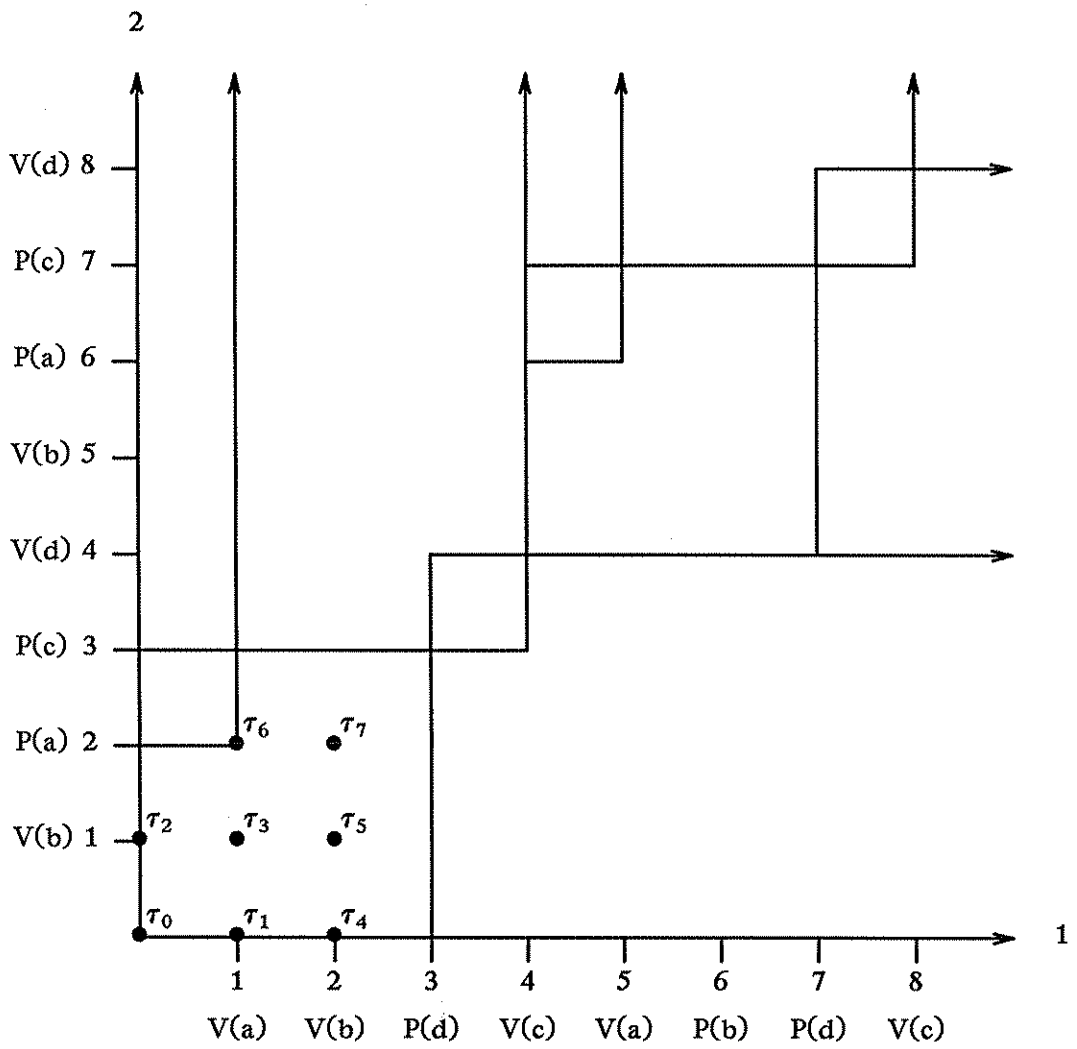


Figure 3-2: *Progress Graph for Example Program.*

The blocking predicate for process 1 is defined by

$$B_1 = (pre(S_1^3) \wedge d = 0) \wedge F^*(\mathbf{false}).$$

and is satisfied by τ_4, τ_5, and τ_7. The blocking predicate for process 2 is defined by

$$B_2 = [(pre(S_2^2) \wedge a = 0) \vee (pre(S_2^3) \wedge c = 0)] \wedge F^*(\mathbf{false})$$

and is satisfied by τ_2, τ_6, and τ_7. The predicate U_1 is defined by

$$U_1 = \neg B_1 \wedge F^*(\mathbf{false})$$

and is satisfied by τ_0, τ_1, τ_2, and τ_3, and τ_6. The predicate U_2 is defined by

$$U_2 = \neg B_2 \wedge F^*(\mathbf{false})$$

and is satisfied by τ_0, τ_1, τ_3, τ_4, and τ_5. Next, we construct the functional G.

$$\begin{aligned} G(J) = J \\ &\vee wp[S_1^1](((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_1^2](((a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_1^3](((a_1^V - 1 = b_1^V - 1 = d_1^P - 1 = c_1^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_1^4](((a_1^V = b_1^V = d_1^P = c_1^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_2^1](((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_2^2](((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_2^3](((b_2^V - 1 = a_2^P - 1 = c_2^P - 1 = d_2^V) \wedge J) \wedge F^*(\mathbf{false}) \\ &\vee wp[S_2^4](((b_2^V = a_2^P = c_2^P = d_2^V) \wedge J) \wedge F^*(\mathbf{false}) \end{aligned}$$

Next, we construct $G^*(U_1)$. By definition we have $G^0(U_1) = U_1$. Applying G again gives

$$G^1(U_1) = U_1$$

$$\vee\ wp[S_1^1]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge U_1) \wedge F^*(\textbf{false}) \tag{3-14}$$

$$\vee\ wp[S_2^1]((b_2^V - 1 = a_2^P = c_2^P = d_2^V) \wedge U_1) \wedge F^*(\textbf{false}) \tag{3-15}$$

$$\vee\ wp[S_2^2]((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge U_1) \wedge F^*(\textbf{false}) \tag{3-16}$$

Weakest precondition 3-14 is satisfied by τ_0 and τ_2. Weakest precondition 3-15 is satisfied by τ_0 and τ_1. Weakest precondition 3-15 is satisfied by τ_3. Thus

$$G^1(U_1) = U_1 = G^*(U_1).$$

Next, we construct $G^*(U_2)$. By definition we have $G^0(U_2) = U_2$. Applying G again gives

$$G^1(U_2) = U_2$$

$$\vee\ wp[S_1^1]((a_1^V - 1 = b_1^V = d_1^P = c_1^V) \wedge U_2) \wedge F^*(\textbf{false}) \tag{3-17}$$

$$\vee\ wp[S_1^2]((a_1^V - 1 = b_1^V - 1 = d_1^P = c_1^V) \wedge U_2) \wedge F^*(\textbf{false}) \tag{3-18}$$

$$\vee\ wp[S_2^2]((b_2^V - 1 = a_2^P - 1 = c_2^P = d_2^V) \wedge U_2) \wedge F^*(\textbf{false}) \tag{3-19}$$

Weakest precondition 3-17 is satisfied by τ_0 and τ_2. Weakest precondition 3-18 is satisfied by τ_1 and τ_3. Weakest precondition 3-19 is satisfied by τ_1 and τ_4. Applying G again, we get

$$G^2(U_2) = G^1(U_2) = G^*(U_2)$$

which is satisfied by τ_0, τ_1, τ_2, τ_3, τ_4, and τ_5. D_1 is defined by

$$D_1 = \neg G^*(U_1) \wedge B_1$$

and satisfied by τ_4, τ_5, and τ_7. Notice that τ_4 and τ_5 are partial deadlock states, since process 1 is deadlocked in these states while process 2 is not blocked. D_2 is defined by

$$D_2 = \neg G^*(U_2) \wedge B_2$$

and satisfied by τ_6 and τ_7. Notice that τ_6 is a partial deadlock state, since 2 is deadlocked in this state while process 1 is not blocked. Notice also how we have strengthened the blocking predicate for process 2 by eliminating τ_2, where process 2 is blocked but eventually becomes unblocked in τ_3. The strongest total deadlock predicate is defined by

$$D = D_1 \wedge D_2$$

and is satisfied by total deadlock state τ_7, where both processes are blocked forever. The generalized deadlock predicate is defined by

$$D = D_1 \vee D_2$$

which is satisfied by partial deadlock states τ_4, and τ_5, and total deadlock state τ_7.

3.5. Chapter Summary

We have presented a new technique for constructing a generalized deadlock predicate from the text of a semaphore program such that the satisfiability of the predicate is a necessary and sufficient condition for the existence of a reachable total or partial deadlock state. Our method is based on Clarke's method for generating strongest resource invariants.

Our method constructs a strongest resource invariant, $F^*(\mathbf{false})$ using Clarke's fixed point functional. We then use $F^*(\mathbf{false})$ and a predicate transformer G^* to construct the generalized deadlock predicate.

As we have defined it, the generalized deadlock predicate can be constructed in practice only for semaphore programs with a finite number of reachable states. However, there exist useful semaphore programs with an infinite number of reach-

able states. For these programs the strongest resource invariant $F^*(\mathbf{false})$ cannot be obtained through a finite number of applications of the fixed point functional, and since the generalized deadlock predicate uses the strongest resource invariant, the generalized deadlock predicate cannot be constructed either. However, some recent work by Carson in [CARS84] suggests that there might be a way to construct a generalized deadlock predicate for a semaphore program with an infinite number of reachable states.

Carson has shown in [CARS84] that we only need to analyze a finite subset of the potentially infinite set of states in a program to accurately predict if the program can deadlock. Determining the size of this finite subset of states is called the *finite models problem.* The finite models problem for general semaphore programs is open.

That a finite subset of states is always sufficient suggests that it is unnecessary to obtain the strongest resource invariant in order to perform accurate deadlock detection with the generalized deadlock predicate. Rather we need only iteratively apply the fixed point functional until we obtain an approximate strongest resource invariant $\hat{F}^*(\mathbf{false})$ that admits a sufficiently large, but finite subset of states. Thus, if we can solve the finite models problem for general semaphore programs, then we also solve the problem of generating the generalized deadlock predicate for general semaphore programs. We address the finite models problem in the next chapter.

CHAPTER 4

Finite Models

Before we can build a geometric model of a semaphore program, we must somehow determine which finite subset of the potentially infinite set of forbidden regions will comprise the model. This problem is called the *finite models problem.* It was identified by Carson in [CARS84] and it is open for general semaphore programs.

In this chapter, we use theory from Petri net models and algebraic models to solve the finite models problem for a subclass of consumable resource semaphore programs. While we have not solved the finite models problem for general semaphore programs, we have increased the number of programs for which a solution is known.

4.1. Finite Geometric Models

A progress graph is a continuous N-dimensional space comprising an infinite set of points that represent program states. In general, a geometric model is a potentially infinite set of potentially infinite forbidden regions scattered throughout the progress graph. This set of forbidden regions is called an *infinite* model. In practice, a geometric model is a finite set of finite forbidden regions. This set of forbidden regions is called a *finite* model.

We encounter two problems when we build finite models. First, we must decide which forbidden regions of the infinite model will comprise the finite model. Second, if any region in the finite model has an infinite extent coordinate in the infinite model, then we must find some finite value to assign to that extent coordinate. This finite value must be chosen such that analysis of the model yields correct conclusions about the behavior of the program.

Given a finite set of forbidden regions, the second problem has been solved by Carson in [CARS84], with his notion of a point representing "effective" infinity and his rule for determining its location (Section 2.3.2). The difficult problem is deciding which forbidden regions to include in the finite model. The finite model must be sufficiently large to yield correct conclusions about the behavior of the program. What we mean by sufficiently large depends upon the conclusions we want to make about the program.

If we simply want to determine that a program has no reachable deadlock states, then we want to build a finite model such that there is a reachable point contained in a deadlock region generated from the forbidden regions of the infinite model if and only if there is a reachable point contained in a deadlock region generated from the forbidden regions of the finite model. Such a finite model is called an *accurate* model.

If, in addition, we are interested in determining all of the processes in a program that can potentially deadlock, then we want to build a finite model such that process i is in the blocked set of a reachable point contained in a deadlock region generated from the forbidden regions of the infinite model if and only if process i is in the blocked set of a reachable point contained in a deadlock region generated

from the forbidden regions of the finite model. Such a finite model is called a *complete* model.

Notice that a complete model is an accurate model, while an accurate model is not necessarily a complete model. We will see an example of this later in the chapter.

While a finite model can be any collection of forbidden regions, we will restrict our attention to finite models constructed by "unfolding" the first k cycles of each process, $k > 0$. The k-cycle subset of a progress graph is the finite N-dimensional space defined by the first k cycles of each process. A k-cycle model is the set of forbidden regions whose vertices are contained within the k-cycle subset of the progress graph.

The problem of building accurate or complete finite models of general semaphore programs is open. However, there have been some preliminary results. Carson has shown that every semaphore program has a complete model[CARS84]. Since every complete model is accurate, every semaphore program has an accurate model as well. Carson has also shown that 1-cycle models are complete for reusable resource programs[CARS84]. This implies that 1-cycle models of reusable resource programs are accurate as well. The remainder of this chapter develops a similar result for a subclass of consumable resource programs called *SS* programs.

4.2. SS Programs and Marked Graphs

In this section we introduce a subclass of consumable resource semaphore programs called *SS* (**S**ingle release **S**ingle request) programs, show how these programs can be modeled with a subclass of Petri nets called marked graphs, and discuss

some of the known properties of marked graphs.

Definition 4-1: SS Programs

A semaphore program is an *SS* program if for each semaphore σ, there is exactly one statement that requests σ and exactly one statement that releases σ. Furthermore these statements are in different processes.

□

An *SS* program is shown in Figure 4-1. We argue that Petri net models of *SS* programs are *marked graphs* [COMM71, HOLT70].

Definition 4-2: Marked Graph

A Petri net N is a marked graph if every place has exactly one input transition and exactly one output transition.

□

By the construction of Petri net models of semaphore programs outlined in Section 2.3.2, we note that each program counter place has exactly one input transition and exactly one output transition, regardless of the semaphore program being modeled. Also by the construction, we see that if k statements release semaphore σ, then the corresponding semaphore place in the Petri net model has exactly k input transitions. Similarly, if k statements request semaphore σ, then the corresponding semaphore place in the Petri net model has exactly k output transitions. Since each semaphore in an *SS* program is released by exactly one statement and requested by exactly one statement, we see that every semaphore place in the Petri net model of an *SS* program has exactly one input transition and one output transition. Hence, all places in the Petri net model of an *SS* program have exactly one input transition and exactly one output transition, which satisfies the definition of a marked

```
a,b,c = 0: semaphore;
cobegin
        1: cycle
                P(a);
                V(b)
           endcycle
        //
        2: cycle
                V(c);
                P(b);
                V(a)
           endcycle
        //
        3: cycle
                P(c)
           endcycle
coend
```

Figure 4-1: *An SS Program.*

graph. The marked graph model of the *SS* program of Figure 4-1 is shown in Figure 4-2.

Some of the first non-trivial results for Petri nets were obtained for marked graphs in [COMM71, GENR73, HOLT70].

P1: The token count of a circuit in a marked graph is invariant over transition firings.

P2: A marking M of a marked graph is live if and only if every circuit is marked in M.

P3: A marking M of a marked graph is live if and only if there are no dead transitions in M.

P4: The number of times that a non-live transition t in a marked graph can fire from a marking M before it becomes dead is a constant, denoted $d(M,t)$ and determined by the minimum token count along all paths from transitions that are dead in M to t. If there is no path from a transition that is dead in M to t, then t is live in M and $d(M,t) = \infty$.

P5: If marking $M' = M + \overline{w}$ of a marked graph is reachable from M, then transition t is dead in M if and only if $\Psi(w)(t) = d(M,t)$. This implies that a transition t is dead in M if and only if $d(M,t) = 0$.

In Figure 4-2, the circuit $c = t_1 p_2 t_2 p_9 t_4 p_5 t_5 p_8 t_1$ is unmarked in M_0. By property $P2$, M_0 is not a live marking, which by property $P3$ implies the existence of transitions that are dead in M_0. By property $P4$, $d(M_0,t_i) = 0$ for

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_1	P(a)	t_3	V(c)	t_6	P(c)	P_7	a
t_2	V(b)	t_4	P(b)			P_8	b
		t_5	V(a)			P_9	c

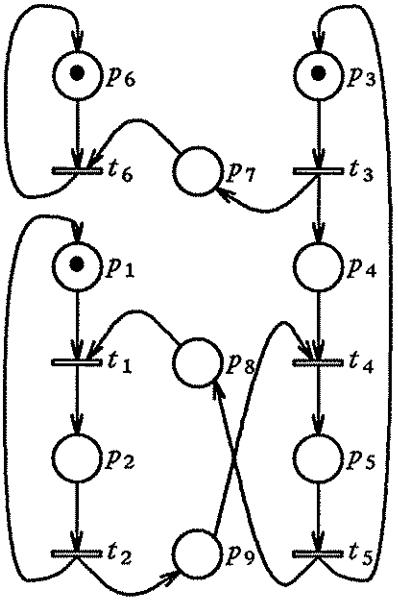


Figure 4-2: *Marked Graph Petri Net Model of an SS Program.*

$i \in \{1,2,4,5\}$ and $d(M_0,t_i) = 1$ for $i \in \{3,6\}$. By property $P5$, transitions t_1, t_2, t_4, and t_5 are dead in M_0.

We will use these known properties of marked graphs to help us derive accurate and complete geometric models of SS programs in the following sections. We will also use the straightforward mapping that exists between semaphore programs and their Petri net models.

Recall from Section 2.4.2 that for a semaphore program C with initial state τ_0 and its Petri net model N with initial marking M_0, a state τ of C corresponds to some marking M of N. The initial state τ_0 of C corresponds to the initial marking M_0 of N. A statement S in C corresponds to some transition t in N.

Statement S is dead in state τ if and only transition t is dead in marking M; statement S is live in state τ if and only if transition t is live in marking M. A sequence of statements $w = S_1S_2 \cdots S_k$ is executable in state τ if and only transition sequence $v = t_1t_2 \cdots t_k$ is enabled in marking M and for all i, $S_i \in w$ corresponds to $t_i \in v$.

State τ' is reachable from state τ if and only if marking M' is reachable from marking M. State τ is a total deadlock state if and only if marking M is dead. If state τ is a partial deadlock state then marking M is not live; the converse is not necessarily true. Finally program C is deadlock-free if and only if initial marking M_0 of N is live.

4.3. Accurate Geometric Models of SS Programs

In this section, we show that 1-cycle models of *SS* programs are accurate. First we establish a property of marked graphs.

Lemma 4-1: For a marked graph $N = (\Pi,\Sigma,F,B)$ and a marking M, there exists a sequence $w \in \Sigma$ of length m enabled in M and containing exactly one instance of each of the m transitions that are not dead in M.

Proof: Suppose that the contrary is true. Now suppose that we start with the empty sequence λ and attempt to build w by adding one enabled transition at a time. Then by assumption, we must always arrive at some sequence v of length less than m such that v is enabled in M, and for all t such that $t \notin v$ and t is not dead in M, t is not enabled in marking $M' = M + \bar{v}$.

Since t is not enabled in M', there is some $p \in \bullet t$ that is unmarked in M'. By the definition of marked graphs, there is exactly one t' such that $t' = \bullet p$. We know that $t' \notin v$, since p is unmarked in M', $t \notin v$, and t is the only output transition of p. We also know that p is unmarked in M, since p is unmarked in M', neither t nor t' is in v, and t' and t are the only input and output transitions, respectively, of p. This implies that t' is not dead in M, by property $P4$ of marked graphs and by the fact that t is not dead in M. Thus t' is a transition such that $t' \notin v$ and t' is not dead in M. This implies that t' is not enabled in M'.

We repeat the same argument for t' to get a transition t'' that is not enabled in M'. We repeat the same argument for t'' to get a transition t''' that is not enabled in M', and so on. Eventually we get a circuit c such that c is unmarked in M' and for all transitions $t \in c$, t is not dead in M. But by property $P1$ of marked graphs, this implies that c is unmarked in M, which implies that the transitions in c are dead in M. Contradiction.

□

Lemma 4-1 tells us that there exists a marking M' that can be reached from a marking M by firing each transition that is not dead in M exactly once. For the

marked graph model in Figure 4-2, $w = t_3 t_6$ is such a sequence of transition firings from the initial marking M_0. Lemma 4-1 suggests a useful property of *SS* programs.

Lemma 4-2: If an *SS* program C is not deadlock-free, then there exists a deadlock state τ that is reachable from the initial state τ_0 by executing each statement that is not dead in τ exactly once.

Proof: Let marked graph N with initial marking M_0 be the Petri net model of *SS* program C with initial state τ_0. By Lemma 4-1 there exists a marking M that can be reached from M_0 by firing each transition that is not dead in M_0 exactly once. This implies a state $\tau = (pc_1, \ldots, pc_N; s)$ that can be reached from the initial state τ_0 by executing each statement that is not dead in τ_0 exactly once. By assumption, C is not deadlock-free, which implies that M_0 is not live, which by property $P3$ of marked graphs implies at least one dead transition in M_0, which implies at least one dead statement in τ_0. For each process i in C with a statement that is dead in τ_0, let S_i^k be the statement such that S_i^k is dead in τ_0 and for $1 \leqslant j < k$, S_i^j is not dead in τ_0. Clearly, S_i^k is dead in τ, and since τ is reached by firing each statement that is not dead in τ_0 exactly once, $pc_i = k$. Thus process i is blocked on a statement that is dead in τ, which implies that τ is a deadlock state.

□

For the *SS* program in Figure 4-2, $\tau = (1,2,1; a=b=c=0)$ is a deadlock state that can be reached by executing the $V(c)$ statement in process 2 followed by the $P(c)$ statement in process 3. Lemma 4-2 suggests that 1-cycle geometric models of *SS* programs are accurate.

Theorem 4-1: 1-cycle geometric models of *SS* programs are accurate.

Proof: For an *SS* program C, we show that there exists a reachable point contained in a deadlock region generated from the infinite model if and only if there exists a reachable point contained in a deadlock region generated from the 1-cycle model.

If: Trivial, given a proper location for effective infinity.

Only If: If there exists a reachable point in a deadlock region generated from the infinite model, then C is not deadlock-free, which by Lemma 4-2 implies a reachable deadlock state τ where some set of processes $\{1, \ldots, \zeta\}$ is deadlocked on the first cycle and where each process $k \notin \{1, \ldots, \zeta\}$ has executed each of its statements exactly once. Consider process $i \in \{1, \ldots, \zeta\}$ that is blocked on semaphore σ in τ at a statement which by the definition of *SS* programs is the only statement that requests σ. Then the statement that releases σ must be in some process $j \in \{1, \ldots, \zeta\}$, since each process $k \notin \{1, \ldots, \zeta\}$ has executed each of its statements exactly once. Thus τ corresponds to a reachable point P in the 1-cycle subset of the progress graph. The blocked set associated with P is $B(P) = \{1, \ldots, \zeta\}$ and the invariant set is $I(P) = \{i \mid i \notin B(P)\}$. This implies that P is a reachable point in a deadlock region whose vertex is contained in the 1-subset of the progress graph. Given a proper location for effective infinity, this implies that S is contained in a deadlock region generated from the 1-cycle model.

□

To see how Theorem 4-1 is applied, consider the 1-cycle geometric model of the *SS* program in Figure 4-2. Using Carson's rule for locating effective infinity, we let $\infty = (5,7,3)$. Then the 1-cycle geometric model consists of three forbidden regions.

Forbidden Regions $\infty = (5,7,3)$			
Number	Semaphore	Vertex	Extent
R_1	a	(1,0,0)	(5,3,3)
R_2	b	(0,2,0)	(2,7,3)
R_3	c	(0,0,1)	(5,1,3)

We begin the analysis of the model by generating the nearness regions of degree 1 formed from the forbidden regions.

Nearness Regions of Degree 1 $\infty = (5,7,3)$					
Number	Forbidden Region	Vertex	Extent	Blocked Set	Invariant Set
N_1	R_1	(0,0,0)	(1,3,3)	{1}	{3}
N_2	R_2	(0,1,0)	(2,2,3)	{2}	{3}
N_3	R_3	(0,0,0)	(5,1,1)	{3}	{1}

None of these nearness regions is a deadlock region, so we intersect them to form the nearness regions of degree 2.

Nearness Regions of Degree 2 $\infty = (5,7,3)$					
Nearness Region	Parents (deg 1)	Vertex	Extent	Blocked Set	Invariant Set
N_4	N_1,N_2	(0,1,0)	(1,2,3)	{1,2}	{3}
N_5	N_1,N_3	(0,0,0)	(1,1,1)	{1,3}	$\varnothing$

Region N_4 is a partial deadlock region where processes 1 and 2 are deadlocked and process 3 is not deadlocked. The vertex of region N_4 corresponds to the state where process 1 has completed no statements, process 2 has completed exactly one statement, and process 3 has completed no statements. Inspection of the program text shows that this is a reachable state; hence the program is not deadlock-free.

The 1-cycle geometric model of the *SS* program in Figure 4-2 is an accurate model; from it we can determine whether the program is deadlock-free. However, the 1-cycle geometric model is not a complete model; from it we can not determine that process 3 deadlocks on its second cycle.

4.4. Complete Geometric Models of SS Programs

In this section we investigate complete geometric models of *SS* programs. We use properties of marked graphs and algebraic models to derive a simple rule for determining the size of a complete geometric model of an *SS* program. First we establish a property of marked graphs and their maximal markings.

Definition 4-3: Maximal Marking of a Petri Net

A marking M of a Petri net $N = (\Pi,\Sigma,F,B)$ is maximal if every transition $t \in \Sigma$ is either live or dead in M.

□

For a non-live marking M, let $\text{maxd}(M)$ denote $\max\{d(M,t) \mid d(M,t) \neq \infty\}$. Intuitively, $\text{maxd}(M)$ is the maximum number of times that any non-live transition t can fire from marking M. Clearly for a non-live marking M, there exists some t such that $d(M,t) < \infty$, by properties $P3$ and $P5$ of marked graphs.

Lemma 4-3: For a marked graph $N = (\Pi,\Sigma,F,B)$ and a non-live marking M, there exists a sequence $w \in \Sigma^*$, $M \xrightarrow{w} M'$, such that M' is maximal and for all $t \in \Sigma$, $\Psi(w)(t) \leqslant \text{maxd}(M)$.

Proof: We claim that for all marked graphs such that $\text{maxd}(M) \geqslant m$, there exists a sequence w, $M \xrightarrow{w} M'$, such that for all $t \in \Sigma$,

$$d(M,t) \leqslant m \Longrightarrow \Psi(w)(t) = d(M,t)$$

and

$$d(M,t) > m \Longrightarrow \Psi(w)(t) = m.$$

We use induction on m.

Basis: $m = 0$. The claim is clearly satisfied by $w = \lambda$.

Induction Hypothesis: Suppose that the claim is true for $m = k$.

Induction Step: By the hypothesis there exists a sequence w, $M \xrightarrow{w} M'$, and

by Lemma 4-1 there exists a sequence v, $M' \xrightarrow{v}$, containing exactly one instance of each transition not dead in M'. Thus $M \xrightarrow{wv}$. Now let t be any transition. If $d(M,t) \leqslant k$, then $\Psi(w)(t) = d(M,t)$ by the induction hypothesis and $\Psi(v)(t) = 0$ because t is dead in M'. Thus $\Psi(wv)(t) = k$. If $d(M,t) = k + 1$, then $\Psi(w)(t) = k$ by the induction hypothesis and $\Psi(v)(t) = 1$ by Lemma 4-1. Thus $\Psi(wv)(t) = k + 1 = d(M,t)$. In general, if $d(M,t) \leqslant k + 1$, then $\Psi(wv)(t) = d(M,t)$. Finally, if $d(M,t) > k + 1$, then $\Psi(w)(t) = k$ by the hypothesis and $\Psi(v)(t) = 1$ by Lemma 4-1. Thus $\Psi(wv)(t) = k + 1$, and the claim is true for $m = k + 1$.

Setting $m = \text{maxd}(M)$, we have a transition sequence w, $M \xrightarrow{w} M'$, such that for all $t \in \Sigma$,

$$d(M,t) \leqslant \text{maxd}(M) \Longrightarrow \Psi(w)(t) = d(M,t)$$

and

$$d(M,t) > \text{maxd}(M) \Longrightarrow \Psi(w)(t) = \text{maxd}(M)$$

By property $P5$, M' is maximal and for all $t \in \Sigma$, $\Psi(w)(t) \leqslant \text{maxd}(M)$.

□

Lemma 4-3 tells us that if a marking M of a marked graph is not live, then there exists a maximal non-live marking M' that can be reached from M by firing each transition at most $\text{maxd}(M)$ times. Lemma 4-3 and the known properties of marked graphs suggest some useful properties of *SS* programs and their maximal states.

Definition 4-4: Maximal State of a Semaphore Program

A state τ of a semaphore program is maximal if every statement S of C is either live or dead in τ.

□

Let marked graph N with initial marking M_0 be the Petri net model of *SS* program C with initial state τ_0. First we show that the only reachable maximal deadlock states of an *SS* program are those states where at least one process is

deadlocked during its first cycle.

Lemma 4-4: If τ is a maximal deadlock state of *SS* program C, then τ is reachable only if at least one process is deadlocked in τ during its first cycle.

Proof: We show the contrapositive. Suppose that τ is a maximal deadlock state where every process deadlocked in τ has completed at least one cycle. This implies that no statement is dead in τ_0, which implies that no transition is dead in initial marking M_0 of the Petri net model of C, which by property $P2$ of marked graphs implies that M_0 is live, which implies that C is deadlock free, which implies that C has no reachable deadlock states, which implies that τ is not reachable.

□

Next, we establish a property of statements that are not live in the initial state.

Lemma 4-5: If statement S of an *SS* program C is not live in τ_0, then S is dead in every maximal state reachable from τ_0.

Proof: If S is not live in τ_0, then corresponding transition t in the Petri net model is not live in M_0. If t is not live in M_0, then by property $P4$ of marked graphs, $d(M_0,t) < \infty$. Now suppose that t is live in some marking M' reachable from M_0. This implies that t can be fired an infinite number of times from M', which implies that t can be fired an infinite number of times from M_0, which contradicts the assumption that $d(M_0,t) > \infty$. Thus, t is not live in all markings reachable from M_0, which implies that S is not live in all states reachable from τ_0, which implies that S is dead in all maximal states reachable from τ_0.

□

Finally we establish a property of *SS* programs and their maximal deadlock states.

Lemma 4-6: If an *SS* program C is not deadlock-free, then there exists a maximal deadlock state τ that can be reached from τ_0 by executing each statement of C at most maxd(M_0) times.

Proof: Since C is not deadlock free, marking M_0 of its Petri net model is not live, which by Lemma 4-3 implies a maximal marking M that can be reached from M_0 by firing each transition $t \in \Sigma$ at most maxd(M_0) times. By the direct mapping between semaphore programs and their Petri net models, this implies a maximal state τ that can be reached by executing each statement of C at most maxd(M_0) times. By the assumption that C is not deadlock-free, τ is a maximal deadlock state.

□

Lemmas 4-5 and 4-6 suggest that for an *SS* program C that is not deadlock-free, a maxd(M_0)-cycle geometric model of C is a complete model. Now, we could determine maxd(M_0) by inspecting all of the paths leading to all of the transitions in the Petri net model of C. However, there is a simple way to determine an upper bound on maxd(M_0) given only the number of processes and the initial value of each semaphore. Our derivation uses the algebraic model.

As in [CARS84], we introduce a *repetition* auxiliary variable, R_i, to the preconditions of the statements in each process i. Repetition auxiliary variables count the number of cycles completed by each process. Each R_i is initially zero and is incremented when the last statement in process i completes. Repetition auxiliary variables are used by Carson in [CARS84] to reason about the number of cycles executed by processes in total deadlock states. Our contribution here is to show how this approach can be used to reason about the number of cycles executed by processes deadlocked in a maximal deadlock state, which can be either a total or a partial deadlock state.

Suppose that τ is a maximal deadlock state where $B(\tau) = \{1, \ldots, \zeta\}$ is the set of processes deadlocked in τ at statements $S_1, \ldots, S_\zeta$. For each statement S_i such that S_i is blocked on some semaphore σ and $i \in B(\tau)$, the predicate

$$pre(S_i) \wedge (\sigma = 0)$$

is satisfied in state τ. Let σ be requested by process i and released in process j. Since τ is a maximal deadlock state, we know that $j \in B(\tau)$, and from the definition of *SS* programs that $i \neq j$. If we substitute the repetition auxiliary variables R_i and R_j from $pre(S_i)$ and $pre(S_j)$ into the semaphore invariant for σ, we get an equation of the form

$$\begin{aligned} \sigma = 0 &= \sigma_0 + \sigma_j^V - \sigma_i^P \\ &= \sigma_0 + R_j + c_i' - R_i \ , i \neq j \end{aligned}$$

where c_i' is a constant zero if σ is released after S_j in the text of process j, and unity if σ is released before S_j in the text of process j. Rearranging terms gives

$$R_i = R_j + c_i \ , i \neq j$$

where $c_i = \sigma_0 + c_i'$. This equation, called a *cyclic dependency equation*, describes the cyclic relationship between deadlocked process i and deadlocked process j in maximal deadlock state τ. If we perform a similar substitution for all of the ζ processes in $B(\tau)$, we get a system of ζ cyclic dependency equations in ζ unknowns of the form:

$$R_1 = R_{i_1} + c_1, \quad i_1 \neq 1 \in \{1, \ldots, \zeta\}$$

$$R_2 = R_{i_2} + c_2, \quad i_2 \neq 2 \in \{1, \ldots, \zeta\}$$

$$\ldots$$

$$R_\zeta = R_{i_\zeta} + c_\zeta, \quad i_\zeta \neq \zeta \in \{1, \ldots, \zeta\}$$

where $0 \leqslant c_i \leqslant \max(\sigma_0) + 1$, and where $\max(\sigma_0)$ denotes the largest initial semaphore value.

The system of cyclic dependency equations associated with a maximal deadlock state τ of an *SS* program provides some insight into the execution history of processes deadlocked in τ: at least two of the processes deadlocked in τ have completed the same number of cycles and all other processes are deadlocked at later cycles.

Lemma 4-7: If τ is a maximal deadlock state, and $\{1, \ldots, \zeta\}$ is the set of processes deadlocked in τ, then for any solution to the system of cyclic dependency equations associated with τ,

$$\exists_{i \neq j \in \{1 \ldots \zeta\}} : R_i = R_j \tag{4-1}$$

and

$$\forall_{k \in \{1 \ldots \zeta\}} : R_i \leqslant R_k . \tag{4-2}$$

Proof: Suppose that Equation 4-1 is not true, such that

$$\forall_{i \in \{1 \ldots \zeta\}} : c_i > 0.$$

This assumption leads, after some renumbering of subscripts, to an equation of the form

$$R_1 = R_2 + c_1 = \cdots = R_k + c_{k-1} = R_\xi + c_k, \quad 1 \leqslant \xi \leqslant k$$

where $i \leqslant k$ implies $c_i \leqslant c_k$. Contradiction. To show that Equation 4-2 is true, we partition the system of equations into disjoint sets A and B with

$$A = \{R_i \mid R_i = R_j\}$$

$$B = \{R_i \mid R_i = R_j + c_i\}, \quad c_i > 0$$

and let $R_m = \min(A)$. If $B = \varnothing$ we are done. If $B \neq \varnothing$, then consider some $R_1 \in B$ with $R_1 = R_2 + c_1$. If $R_2 \in A$ then

$$R_1 > R_2 \geqslant R_m .$$

On the other hand, if $R_2 \in B$ then $R_2 = R_3 + c_2$ and

$$R_1 > R_2 > R_3.$$

Similarly, if $R_3 \in B$ then $R_3 = R_4 + c_3$ and

$$R_1 > R_2 > R_3 > R_4.$$

Continuing in this fashion, we must encounter some $R_i \in A$ such that

$$R_1 > R_2 > R_3 > R_4 > \cdots > R_i \geqslant R_m.$$

Thus $R_m < \min(B)$, which establishes the truth of Equation 4-2.

□

With this result and Lemma 4-4 we can determine an upper bound on the number of cycles completed be any process deadlocked in a reachable maximal deadlock state.

Lemma 4-8: If τ is a reachable maximal deadlock state of *SS* program *C* then each process $i \in = \{1, \ldots, \zeta\}$ deadlocked in τ has completed at most $(N-2)(\max(\sigma_0)+1)$ cycles.

Proof: By Lemma 4-7 we know that at least two processes are deadlocked in the same cycle and no other processes are deadlocked in an earlier cycle. Thus, the largest difference in the number of cycles completed by the processes deadlocked in τ is given by an equation of the form:

$R_1 = R_2 + \max(\sigma_0) + 1$

$R_2 = R_3 + \max(\sigma_0) + 1$

$\ldots$

$R_{\zeta-2} = R_{\zeta-1} + \max(\sigma_0) + 1$

$R_{\zeta-1} = R_\zeta$

$R_\zeta = R_{\zeta-1}$

By Lemma 4-4 and the assumption that τ is reachable, at least one process is deadlocked in its first cycle. Thus $R_\zeta = 0$ and

$$R_1 = (\zeta-2)(\max(\sigma_0)+1) \leqslant (N-2)(\max(\sigma_0)+1).$$

Hence, no process that is deadlocked in τ has completed more than $(N-2)(max(\sigma_0)+1)$ cycles.

□

If τ is a reachable maximal deadlock state, then by Lemma 4-8, no process deadlocked in τ has completed more than $(N-2)(\max(\sigma_0)+1)$ cycles. This implies that no statement that is dead in τ has executed more than $(N-2)(\max(\sigma_0)+1)+1$ times. If we let $K = \max(\sigma_0)+1$, then no statement that is dead in τ has executed more than $NK-2K+1$ times. Recalling that $\text{maxd}(M_0)$ is the maximum number of times that any non-live statement can execute from τ_0, we have $\text{maxd}(M_0) \leqslant NK-2K+1$, which suggests that $(NK-2K+1)$-cycle geometric models of *SS* programs are complete.

Theorem 4-2: $(NK-2K+1)$-cycle geometric models of *SS* programs are complete.

Proof: We show that a process i is in the blocked set of a reachable point contained in a deadlock region generated from the infinite model if and only if process i is in the blocked set of a reachable point contained in a deadlock region generated from the (NK - 2K + 1)-cycle model.

If: Trivial given a proper value for effective infinity.

Only If: If process i is in the blocked set of a reachable point contained in a deadlock region generated from the infinite model, then the program is not deadlock-free. By Lemma 4-6, there exists a maximal deadlock state τ that can be reached from the initial state by executing no statement more than $\text{maxd}(M_0)$ times, which by Lemma 4-8 implies that τ can be reached from τ_0 by executing no statement more than $NK-2K+1$ times. By Lemma 4-5, process i is deadlocked in τ. Furthermore, for each $j \in \{1, \ldots, \zeta\}$ deadlocked on semaphore σ in τ, there is no process $k \notin \{1,...,\zeta\}$ that releases σ by the assumption that τ is maximal.

Thus, there is a reachable point P corresponding to τ in the $(NK-2K+1)$-cycle subset of the progress graph where the blocked set of P is $B(P) = \{1, \ldots, \zeta\}$, the invariant set is $I(P) = \{i \mid i \notin B(P)\}$, and process $i \in B(P)$. This implies that process i is in the blocked set of a reachable point in a deadlock region whose vertex is contained in the $(NK-2K+1)$-cycle subset of the progress graph. Given a proper location for effective infinity, this implies that process i is in the blocked set of a reachable point in a deadlock region generated from the $(NK-2K+1)$-cycle model.

□

For the example *SS* program in Figure 4-2, $NK - 2K + 1 = 3(1) - 2(1) + 1 = 2$. To see how Theorem 4-2 is applied, consider the 2-cycle geometric model of the example *SS* program. Using the rule for locating effective infinity, we let $\infty = (7,10,4)$. Then the 2-cycle geometric model consists of six forbidden regions.

Forbidden Regions $\infty = (7,10,4)$			
Number	Semaphore	Vertex	Extent
R_1	a	(1,0,0)	(7,3,4)
R_2	b	(0,2,0)	(2,10,4)
R_3	c	(0,0,1)	(7,1,4)
R_4	a	(3,3,0)	(7,6,4)
R_5	b	(2,5,0)	(4,10,4)
R_6	c	(0,1,2)	(7,2,4)

We begin the analysis of the model by generating the nearness regions of degree 1 formed from the forbidden regions.

Nearness Regions of Degree 1 $\infty = (7,10,4)$					
Number	Forbidden Region	Vertex	Exent	Blocked Set	Invariant Set
N_1	R_1	(0,0,0)	(1,3,4)	{1}	{3}
N_2	R_2	(0,1,0)	(2,2,4)	{2}	{3}
N_3	R_3	(0,0,0)	(7,1,1)	{3}	{1}
N_4	R_4	(2,3,0)	(3,6,4)	{1}	{3}
N_5	R_4	(3,2,0)	(7,3,4)	{2}	{1,3}
N_6	R_5	(1,5,0)	(2,10,4)	{1}	{2,3}
N_7	R_5	(2,4,0)	(4,5,4)	{2}	{3}
N_8	R_6	(0,0,2)	(7,1,4)	{2}	{1,3}
N_9	R_6	(0,1,1)	(7,2,2)	{3}	{1}

Regions N_5, N_6, and N_8 are deadlock regions. Process 3 is not deadlocked in any of these regions, so we intersect them to form the nearness regions of degree 2.

Nearness Regions of Degree 2 $\infty = (7,10,4)$					
Nearness Region	Parents (deg 1)	Vertex	Extent	Blocked Set	Invariant Set
N_{10}	N_1,N_2	(0,1,0)	(1,2,4)	{1,2}	{3}
N_{11}	N_1,N_3	(0,0,0)	(1,1,1)	{1,3}	$\varnothing$
N_{12}	N_4,N_7	(2,4,0)	(3,5,4)	{1,2}	{3}
N_{13}	N_1,N_8	(0,0,2)	(1,1,4)	{1,2}	{3}
N_{14}	N_1,N_9	(0,1,1)	(1,2,2)	{1,3}	$\varnothing$
N_{15}	N_2,N_9	(0,1,1)	(2,2,2)	{2,3}	$\varnothing$

Regions N_{10}, N_{12}, and N_{13} are deadlock regions. Process 3 is not deadlocked in any of these regions, so we intersect them to form nearness regions of degree 3.

Nearness Regions of Degree 3 $\infty = (7,10,4)$					
Nearness Region	Parents (deg 2)	Vertex	Extent	Blocked Set	Invariant Set
N_{16}	N_1,N_2,N_9	(0,1,1)	(1,2,2)	{1,2,3}	$\varnothing$

Region N_{16} is a deadlock region where all three processes are deadlocked. The vertex of region N_{16} corresponds to the state where process 1 has completed no statements, process 2 has completed exactly one statement, and process 3 has completed exactly one statement. Inspection shows that this is a reachable point. Thus, the 2-cycle model is a complete model.

4.5. Chapter Summary

We have used theory from Petri net models and algebraic models to solve the finite models problem for a subclass of consumable resource semaphore programs called *SM* programs. We have introduced the notions of accurate and complete models and shown how the sizes of these models can be determined from the text of a program.

The finite models problem is an interesting and important area for future research, for without a solution, the geometric model is not a practicable tool for the analysis of general semaphore programs.

CHAPTER 5

Reachability Results for a Subclass of Petri Nets

In the previous chapter we solved the finite models problem for a subclass of consumable resource semaphore programs, in the hope of gaining some insight into the as yet unsolved finite models problem for general semaphore programs. If we fail to find a general solution, or if the general solution is too complex, then we would like to identify subclasses of semaphore programs for which the finite models problems can be solved efficiently.

Similar reasoning was evident in the Petri net literature during the late sixties and seventies with respect to the reachability problem. For many years, the decidability of the reachability problem was unknown. Researchers investigated subclasses of Petri nets, and in the process discovered subclasses of Petri nets whose reachability sets had a property called *semilinearity* ([LAND78] for example). It was found that semilinearity was a strong indication of decidable reachability. These observations led eventually to the general solution of the reachability problem developed by E. Mayr in [MAYR81, MAYR84].

The complexity and practicality of Mayr's algorithm are unknown at this writing. Thus, we are motivated to find subclasses of Petri nets for which reachability can be decided in a simpler way. In this chapter, we present a new subclass of Petri nets called *bounded inverse* Petri nets for which reachability is decidable by a straightforward enumeration of a finite set of markings, even though the reachability sets of these nets are potentially infinite. This result allows us to

show how bounded inverse nets can be used to model general semaphore programs.

5.1. Subclasses of Petri Nets

Subclasses of Petri nets are traditionally defined in terms of *static* properties or in terms of *dynamic* properties[PETE81]. A static subclass is defined by making restrictions on the Petri net $N = (\Pi,\Sigma,F,B)$. A dynamic subclass is defined by making restrictions on the markings in the reachability set $R(N,M)$. We present the subclasses of Petri nets that we have found in our study of the literature. We will use these definitions later in the chapter when we argue that bounded inverse nets are a new subclass of Petri nets.

The dynamic subclasses of Petri Nets include bounded nets[KARP69], persistent nets[GRAB80, LAND78], semi-persistent nets[YAMA81], reversible nets[ARAK77], and non forced-choice nets[MEMM78].

Definition 5-1: Bounded Net(B)

A place $p \in \Pi$ of a Petri net $N = (\Pi,\Sigma,F,B)$ is bounded in marking M if and only if there exists some k such that for all $M' \in R(N,M)$, $M'(p) \leqslant k$. A Petri net $N = (\Pi,\Sigma,F,B)$ is bounded in marking M if and only if for all places $p \in \Pi$, p is bounded is bounded in M.

□

Definition 5-2: Persistent Net(P)

A Petri net $N = (\Pi,\Sigma,F,B)$ with initial marking M is persistent if and only if for all $t \neq t' \in \Sigma$ and for all $M' \in R(N,M)$, $M' \xrightarrow{t}$ and $M' \xrightarrow{t'} \Rightarrow M' \xrightarrow{tt'}$.

□

Definition 5-3: Weakly Persistent Net(WP)

A Petri net $N = (\Pi,\Sigma,F,B)$ with initial marking M is weakly persistent if and only if for all $M' \in R(N,M)$, for all $t \in \Sigma$, and for all $w \in \Sigma^*$, $M' \xrightarrow{t}$ and $M' \xrightarrow{wt} \Rightarrow M' \xrightarrow{tv}$ for some rearrangement v of w.

□

Definition 5-4: Reversible Net(RV)

A Petri net $N = (\Pi,\Sigma,F,B)$ is reversible if and only if for all markings M and M',

$$M \xrightarrow{*} M' \Rightarrow M' \xrightarrow{*} M.$$

□

Definition 5-5: Non Forced-Choice Net(CNI, graphe a choix non impose)

If a Petri net $N = (\Pi,\Sigma,F,B)$ is a non forced-choice net, then for all markings M, for all $t \in \Sigma$, and for all $t' \in (\bullet t)\bullet$, there exists some $w \in \Sigma^*$ such that

$$M \xrightarrow{t} \Rightarrow M \xrightarrow{wt'}$$

□

The inclusion relationships among the dynamic subclasses of Petri nets are shown in Figure 5-1.

The static subclasses of Petri nets include state machines, marked graphs[COMM71, HOLT70, MURA77], forward conflict-free nets[CRES75, HOLT70, LAND78], backward conflict-free nets[LIEN76], free-choice nets[HACK72, THIA84], simple

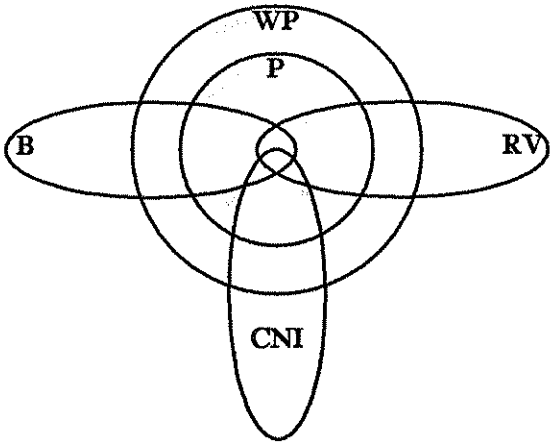


Figure 5-1: *Dynamic Subclasses of Petri Nets.*

nets[COMM72, HACK72], extended simple nets[HOLT74], normal nets[YAMA84], regular nets[DATT84], control structure nets[HERZ77, HERZ79], and non self-controlling nets[GRIE79].

Definition 5-6: State Machine(M)

A Petri net $N = (\Pi, \Sigma, F, B)$ is a state machine if and only if for all transitions $t \in \Sigma$,

$$|\bullet t| = 1 \text{ and } |t \bullet| = 1.$$

□

In words, each transition in a state machine has exactly one input place and exactly one output place.

Definition 5-7: Marked Graph(MG)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a marked graph if and only if for all places $p \in \Pi$,

$$|\bullet p| = 1 \text{ and } |p\bullet| = 1.$$

□

In words, each place in a marked graph has exactly one input transition and exactly one output transition.

Definition 5-8: Forward Conflict-Free Net(CF)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a forward conflict-free net if and only if for all places $p \in \Pi$, $|p\bullet| \leqslant 1$

□

In words, every place in a forward conflict-free net has at most one output transition. Forward conflict-free nets are often referred to as simply *conflict free nets*.

Definition 5-9: Backward Conflict-Free Net(BF)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a backward conflict-free net if and only if for all places $p \in \Pi$, $|\bullet p| \leqslant 1$

□

In words, every place in a backward conflict-free net has at most one input transition.

Definition 5-10: Free-Choice Net(FC)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a free-choice net if and only if for all places $p \in \Pi$,

$$|p\bullet| > 1 \Rightarrow \bullet(p\bullet) = \{p\}.$$

□

In words, if a place in a free-choice net is an input place for a set of transitions, then it is the only input place of those transitions.

Definition 5-11: Non Self-Controlling Net(NSK, nicht selbst kontrollierend)

A place p is a conflict place if $|p\bullet| > 1$. A conflict place p is self-controlling if there is an output transition $t \in p\bullet$ such that there is a simple path from p through t to a $t' \in p\bullet - \{t\}$ and there is a circuit containing p and t. A Petri net $N = (\Pi,\Sigma,F,B)$ is a non self-controlling net if and only if no conflict place of N is self-controlling.

□

Definition 5-12: Simple Net(S)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a simple net if and only if for all transitions $t \in \Sigma$,

$$|\{p \in \bullet t \mid |p\bullet| > 1\}| \leqslant 1.$$

□

In words, a transition in a simple net has at most one input place that is shared with another transition.

Definition 5-13: Extended Simple Net(ES)

A Petri net $N = (\Pi,\Sigma,F,B)$ is an extended simple net if and only if for all places $p,p' \in \Pi$,

$$p\bullet \cap p'\bullet \neq \emptyset \Rightarrow p\bullet \subseteq p'\bullet \vee p'\bullet \subseteq p\bullet$$

□

In words, if two places in an extended simple net are allowed to share output transition only if the set of output transitions of one of the places is a subset of the set of output transitions of the other place.

In [YAMA84], Yamasaki gives both static and dynamic definitions for normal Petri nets. We give one of the static definitions.

Definition 5-14: Normal Net(N)

Let C be the set of transitions and places comprising any minimal circuit of a Petri net $N = (\Pi,\Sigma,F,B)$, and let $p,p' \in \Pi$. Then N is a normal net if for all transitions $t \in \Sigma$,

$$p \in \bullet t \text{ and } p \in C \Rightarrow \exists p' : p' \in C \text{ and } p' \in t\bullet .$$

□

In words, every transition in a normal net with an input place in a minimal circuit has an output place in that minimal circuit.

The remaining two static subclasses, regular nets and control structure nets, have rather complex definitions. For our purposes, it suffices to give necessary conditions for a Petri net to be a regular net or a control structure net.

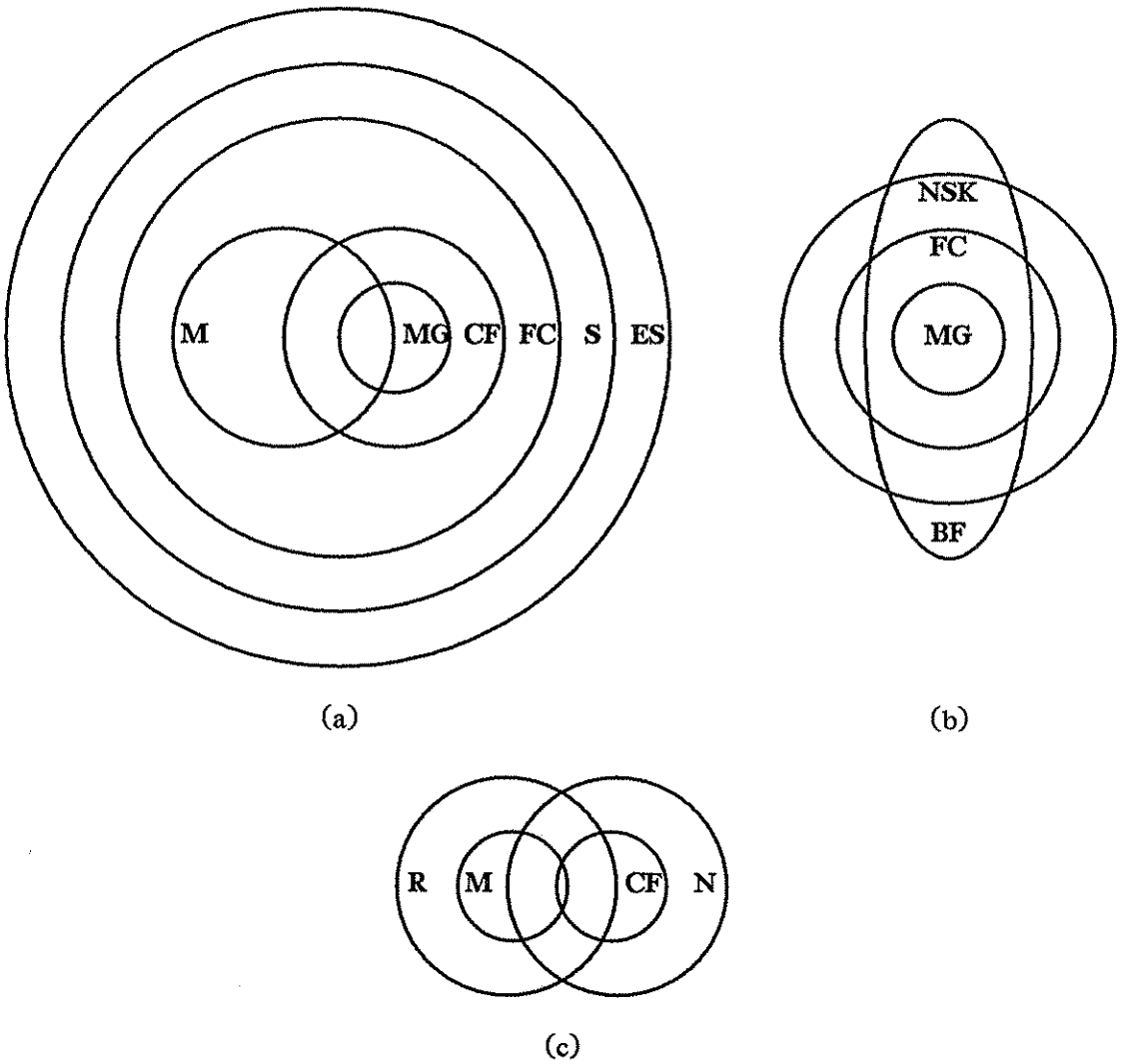


Figure 5-2: *Static Subclasses of Petri Nets.*

Definition 5-15: Regular Net(R)

If a Petri net is a regular net, then it is bounded.

□

Definition 5-16: Control Structure Net(CS)

If a Petri net is a control structure net, then there exists exactly one place that has no input transitions.

□

The inclusion relationships among the static subclasses of Petri nets are shown in Figures 5-2(a), (b), and (c). Control structure nets are not shown because they are incomparable with the other subclasses.

5.2. A Petri Net Reachability Theorem

In this section, we derive a necessary and sufficient condition for a marking M' of a Petri net N to be reachable from a marking M. We begin with the notion of an inverse Petri net[PETE81].

Definition 5-17: Inverse Petri Net

The inverse of a Petri net $N = (\Pi,\Sigma,F,B)$ is a Petri net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$, where $\overline{F}(t) = B(t)$ and $\overline{B}(t) = F(t)$ for each transition $t \in \Sigma$.

□

Next, we show that a marking M' of net N is reachable from a marking M of net N if and only if marking M of the inverse net $\overline{N}$ is reachable from marking M' of the inverse net $\overline{N}$.

Let $N = (\Pi,\Sigma,F,B)$ be a Petri net whose inverse is $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$, let M_N denote a marking of net N, and let $M_{\overline{N}}$ denote a marking of net $\overline{N}$, with $M_N = M_{\overline{N}}$.

Lemma 5-1: $M_N \xrightarrow{t} M_N'$ if and only if $M_{\overline{N}}' \xrightarrow{t} M_{\overline{N}}$.

Proof:

If: Assume that $M_{\overline{N}}' \xrightarrow{t} M_{\overline{N}}$. Then by definition

$$M_{\overline{N}}' \geqslant \overline{F}(t), \tag{5-1}$$

and

$$M_{\overline{N}} = M_{\overline{N}}' - \overline{F}(t) + \overline{B}(t), \tag{5-2}$$

To show that $M_N \xrightarrow{t} M_N'$, we must show that

$$M_N \geqslant F(t) \tag{5-3}$$

and that

$$M_N - F(t) + B(t) = M_N'. \tag{5-4}$$

From 5-1 and 5-2 we have $M_{\overline{N}} \geqslant \overline{B}(t)$, which from the definition of $\overline{N}$ implies $M_{\overline{N}} \geqslant F(t)$. Now, since $M_N = M_{\overline{N}}$, we have $M_N \geqslant F(t)$. Thus 5–3 is true. From 5-2 and the definition of $\overline{N}$ we have

$$\begin{aligned} M_N - F(t) + B(t) &= M_N + \overline{F}(t) - \overline{B}(t) \\ &= M_{\overline{N}} + \overline{F}(t) - \overline{B}(t) \\ &= M_{\overline{N}}' \\ &= M_N'. \end{aligned}$$

Thus 5-4 is true and we have $M_N \xrightarrow{t} M_N'$.

Only If: The proof is symmetric.

□

The result of Lemma 5-1 is easily extended to include sequences of transition firings. Let $w \in \Sigma^*$ be a firing sequence $t_1 t_2 \cdots t_k$ and let ν be rearrangement

$t_k t_{k-1} \cdots t_1$.

Theorem 5-1: $M_N \xrightarrow{w} M_N'$ if and only if $M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$.

Proof: The proof is by induction on the length of w.

Basis: For w of length 1 see Lemma 5-1.

Induction Hypothesis: Suppose that $M_N \xrightarrow{w} M_N'$ if and only if $M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$ for w of length k.

Induction Step: We must show that $M_N \xrightarrow{w} M_N' \xrightarrow{t} M_N''$ if and only if $M_{\overline{N}}'' \xrightarrow{t} M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$.

If: Assume that $M_{\overline{N}}'' \xrightarrow{t} M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$. Then by the induction hypothesis $M_N \xrightarrow{w} M_N'$ and by Lemma 5-1 $M_N' \xrightarrow{t} M_N''$. Thus $M_N \xrightarrow{w} M_N' \xrightarrow{t} M_N''$.

Only If: Assume that $M_N \xrightarrow{w} M_N' \xrightarrow{t} M_N''$. Then by Lemma 5-1 $M_{\overline{N}}'' \xrightarrow{t} M_{\overline{N}}'$ and by the induction hypothesis $M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$. Thus $M_{\overline{N}}'' \xrightarrow{t} M_{\overline{N}}' \xrightarrow{v} M_{\overline{N}}$.

□

Theorem 5-1 was inspired by a technique developed by Carson in [CARS84] to determine the reachability of a point in a geometric model of a semaphore program. Carson observed that the reachability of a point in a geometric model can be decided by working "backwards" from that point to the origin. Theorem 5-1 tells us that a similar property holds for the larger class of systems that can be modeled with Petri nets.

We will use Theorem 5-1 in the next section to identify a new subclass of Petri nets for which reachability is decidable by constructing a reachability tree.

5.3. Bounded Inverse Nets

In this section we introduce a dynamic subclass of Petri nets called *bounded inverse* nets. We argue that there are bounded inverse nets that are not members of any of the subclasses of Petri nets that we have found in our search of the literature. We show that we can decide the reachability of markings of bounded inverse nets by constructing a reachability tree, even though the reachability sets of bounded inverse nets are potentially infinite.

Definition 5-18: Bounded Inverse Net(BI)

A Petri net $N = (\Pi,\Sigma,F,B)$ is a bounded inverse net if its inverse Petri net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$ is bounded in all markings(see Definition 5-1).

□

We argue that there are bounded inverse nets that are not members of any of the subclasses of Petri nets that we have found in our study of the literature. Consider the Petri net N in Figure 5-3(a). We verify that N is a bounded inverse net by showing that the length of every sequence enabled in an arbitrary marking M of $\overline{N}$ is finite and bounded by a linear combination of token counts of places in M.

Let $w \in \Sigma$ be an arbitrary transition sequence that is enabled in marking M of the inverse net $\overline{N}$ shown in Figure 5-3(b). Clearly the following inequalities hold:

$\Psi(w)(t_2) \leqslant M(p_3)$

$\Psi(w)(t_5) \leqslant M(p_7)$

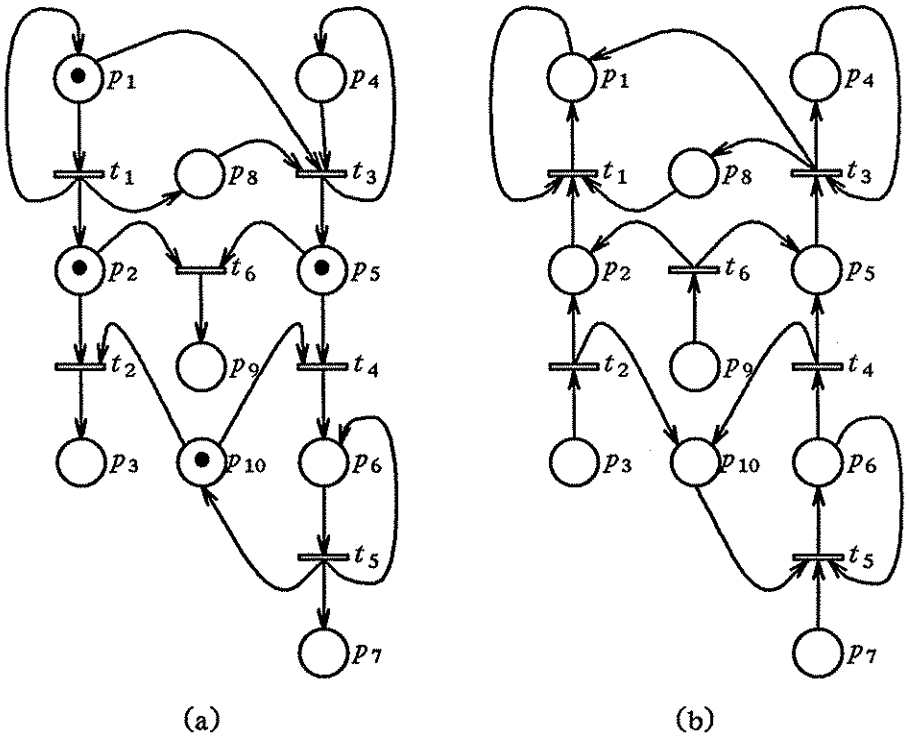


Figure 5-3: *A Bounded Inverse Net.*

$$\Psi(w)(t_6) \leqslant M(p_9).$$

Now, the number of times that transition t_1 can appear in w is bounded by the token count of place p_2 in M, plus the number of times that transition t_2 appears in w, plus the number of times that transition t_6 appears in w. Thus

$$\begin{aligned}\Psi(w)(t_1) &\leqslant M(p_2) + \Psi(w)(t_2) + \Psi(w)(t_6) \\ &\leqslant M(p_2) + M(p_3) + M(p_9).\end{aligned}$$

Similarly we have

$$\Psi(w)(t_4) \leqslant M(p_6) + \Psi(w)(t_5)$$

$$\leqslant M(p_6) + M(p_7),$$

and

$$\begin{aligned}\Psi(w)(t_3) &\leqslant M(p_5) + \Psi(w)(t_6) + \Psi(w)(t_4)\\ &\leqslant M(p_5) + M(p_9) + M(p_6) + M(p_7).\end{aligned}$$

Thus, the length of sequence w is finite and bounded by constant k, which is defined by a linear combination of the form

$$k = \sum_{i=1}^{|\Pi|} c_i M(p_i), \quad 0 \leqslant c_i < \infty.$$

This implies that for a place p and a marking M' reachable from a marking M of the inverse net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$,

$$M'(p) \leqslant M(p) + k$$

which implies that $\overline{N}$ is bounded which implies that N is a bounded inverse net.

Next, we show that the subclass of bounded inverse nets is incomparable with the subclasses of Petri nets defined in Section 5-1. Let $M = (1,1,0,0,1,0,0,0,0,1)$ be the marking of the bounded inverse net $N = (\Pi,\Sigma,F,B)$ shown in Figure 5-3(a).

First we consider whether N belongs to any of the dynamic subclasses of Petri nets.

N is not bounded (Definition 5-1) in M because transition t_1 can be fired an infinite number of times from M.

Refer to the definition of weakly persistent nets (Definition 5-3) and let $t = t_2$, let $w = t_4 t_5$. By inspection we see that $M \xrightarrow{t}$ and $M \xrightarrow{wt}$, yet there is no rearrangement v of w such that $M \xrightarrow{tv}$. Thus N is not weakly persistent, and since

the subclass of weakly persistent nets properly includes the subclass of persistent nets[YAMA81]. N is not persistent, either.

Consider the marking $M' = (1,0,1,0,1,0,0,0,0,0)$ which is reached from the initial marking by firing t_2. Because p_3 never loses tokens as a result of transition firings, there is no sequence w such that $M' \xrightarrow{w} M$. Thus N is not reversible (Definition 5-4).

Refer to the definition of non forced-choice nets (Definition 5-5) and let $t = t_1$, and $t' = t_3$. Clearly $M \xrightarrow{t}$, yet there is no w such that $M \xrightarrow{wt}$. Thus N is not a non forced-choice net(Definition 5-5).

Next we consider whether N is a member of one of the static subclasses of Petri nets.

N is not an extended simple net (Definition 5-13) because $p_2\bullet \cap p_5\bullet = t_6$, yet $p_2\bullet$ and $p_5\bullet$ are incomparable. Since the subclass of simple nets properly includes the subclasses of strongly connected state machines (Definition 5-6), marked graphs (Definition 5-7), conflict-free nets (Definition 5-8), free-choice nets (Definition 5-10), and simple nets (Definition 5-12), N is not one of these nets either.

N is not backward conflict-free because p_6 has two input transitions.

Consider the minimal circuit $c = p_{10}t_4p_6t_5p_{10}$. Transition t_2 has an input place in $c(p_{10})$, but no output place in c. Thus, N is not a normal net (Definition 5-14).

N is not a regular net (Definition 5-15) because it is not bounded. It is not a control structure net (Definition 5-16) because every place has at least one input transition. It is not a non self-controlling net (Definition 5-11) because p_1 is a self-controlling conflict place. Thus we conclude that N is not a member of any of the known subclasses of Petri nets.

We have argued that bounded inverse nets are a new subclass of Petri nets. Next we show that we can decide the reachability of markings of bounded inverse nets by constructing a reachability tree, even though the reachability sets of bounded inverse nets are potentially infinite.

Theorem 5-2: For a bounded inverse net $N = (\Pi,\Sigma,F,B)$, it is decidable whether a marking M' is reachable from a marking M by constructing the reachability tree for marking M' of the inverse net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$.

Proof: Since N is a bounded inverse net, the inverse net $\overline{N}$ is bounded in marking M'. This implies that the reachability set $R(\overline{N},M')$ is finite, which implies that the reachability tree of [KARP69] can be used to decide whether marking M of $\overline{N}$ is reachable from marking M' of $\overline{N}$[PETE81]. If marking M of $\overline{N}$ is reachable from marking M' of $\overline{N}$, then by Theorem 5-1, marking M' of N is reachable from marking M of N. Otherwise by Theorem 5-1, marking M' of N is not reachable from marking M of N.

□

In the next section we give an example showing how the reachability of markings of bounded inverse nets can be decided with the reachability tree.

5.4. Bounded Inverse Nets and Semaphore Programs

Although we know of no simple set of static properties that uniquely identifies the subclass of bounded inverse nets, we can show informally that the

subclass of bounded inverse nets properly includes a useful and interesting subclass of Petri nets that model semaphore programs.

Consider the semaphore program in Figure 5-4. To construct a bounded inverse net Petri net model of this program, we first associate a place/transition pair with each statement and a place with each semaphore, as in Figure 5-5(a). Places p_5 and p_6 represent semaphores a and b respectively. Places p_1 and p_2 represent the current value of the program counter in process 1 while places p_3 and p_4 represent the current value of the program counter in process 2. Transitions t_1 and t_2 represent the $P(a)$ and $V(b)$ statements in process 1, while t_3 and t_4 represent the $V(a)$ and $V(b)$ statements in process 2. The sequential nature of each process is modeled by the directed arcs between places and transitions. Next, we model the effects of the P and V statements on the values of their respective semaphores by adding the directed arcs (p_5,t_1), (t_3,p_5), (t_2,p_6), and (p_6,t_4), as in Figure 5-5(b).

```
a = 0, b = 0: semaphore;

cobegin
        1: cycle
                P(a);
                V(b)
        endcycle
        //
        2: cycle
                V(a);
                P(b)
        endcycle
coend
```

Figure 5-4: *A Semaphore Program.*

Legend					
Process 1		Process 2		Semaphores	
transition	statement	transition	statement	place	semaphore
t_1	P(a)	t_3	V(a)	P_5	a
t_2	V(b)	t_4	P(b)	P_6	b

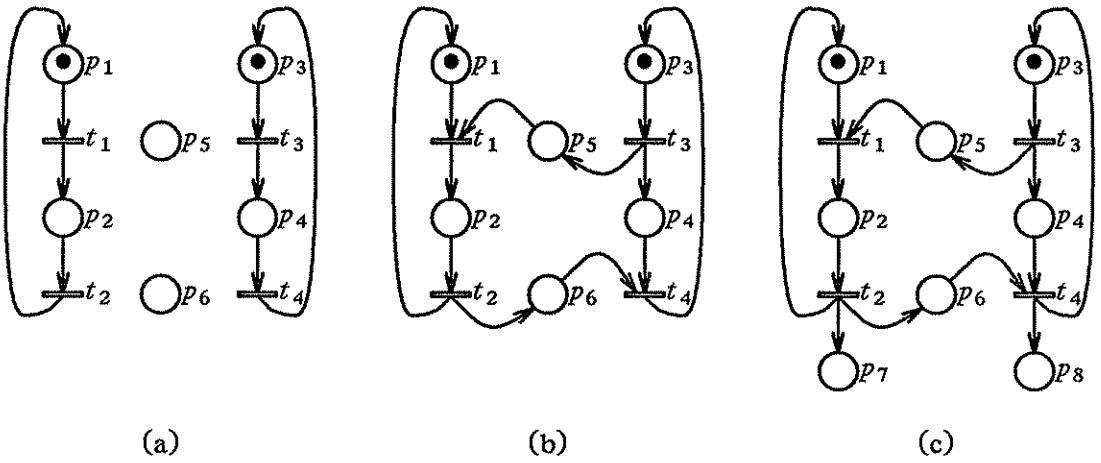


Figure 5-5: *Constructing a Bounded Inverse Petri Net Model of a Semaphore Program.*

The Petri net in Figure 5-5(b) is the standard Petri net model of a semaphore program that we discussed in Section 2.4.2. To create the bounded inverse Petri net model, we we add places p_7 and p_8 and directed arcs (t_2,p_7) and (t_4,p_8), as in Figure 5-2(c). Using this construction, it is clear that we can build a bounded inverse Petri net model of any semaphore program.

Let us confirm that the Petri net of Figure 5-5(c) is a bounded inverse Petri net. The Petri net $N = (\Pi,\Sigma,F,B)$ in Figure 5-5(c) and its inverse net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$ are shown in Figure 5-6.

For any transition sequence $w \in \Sigma$ enabled in initial marking M of $\overline{N}$, it is easy to see that

$$\Psi(w)(t_2) \leqslant M(p_7)$$
$$\Psi(w)(t_4) \leqslant M(p_8).$$

From these inequalities, we can determine that

$$\begin{aligned}\Psi(w)(t_1) &\leqslant M(p_2) + \Psi(w)(t_2)\\ &\leqslant M(p_2) + M(p_7)\end{aligned}$$

and

$$\begin{aligned}\Psi(w)(t_3) &\leqslant M(p_4) + \Psi(w)(t_4)\\ &\leqslant M(p_4) + M(p_8).\end{aligned}$$

Thus, the length of any transition sequence enabled in any marking M is bounded by a constant, which implies that $\overline{N}$ is a bounded net, which implies that N is a bounded inverse net. This argument is easily generalized to arbitrary semaphore programs.

Legend					
Process 1		Process 2		Semaphores	
transition	statement	transition	statement	place	semaphore
t_1	P(a)	t_3	V(a)	P_5	a
t_2	V(b)	t_4	P(b)	P_6	b

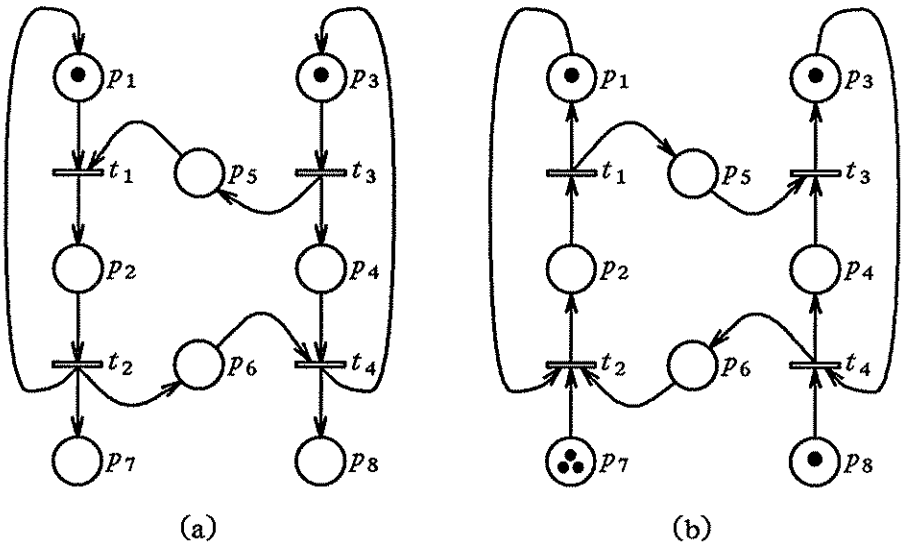


Figure 5-6: *A Bounded Inverse Net and its Inverse.*

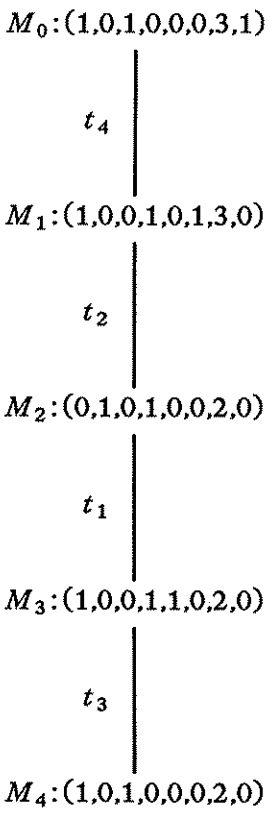


Figure 5-7: *Reachability Tree for Inverse Net.*

To see how the reachability tree can be used to decide the reachability of markings of bounded inverse nets, consider the bounded inverse Petri net model $N = (\Pi,\Sigma,F,B)$ in Figure 5-6(a) with initial marking $M_0 = (1,0,1,0,0,0,0,0)$. We would like know if the marking $M = (1,0,1,0,0,0,3,1)$ is reachable from M_0. Marking M corresponds a state where each process is ready to execute its first statement, with process 1 having completed three cycles and process 2 having completed 1 cycle. To decide reachability we build the inverse net $\overline{N} = (\Pi,\Sigma,\overline{F},\overline{B})$ with initial marking M, as shown in Figure 5-6(b). Next, we build the reachability tree for $\overline{N}$ shown in Figure 5-7. Inspection of the tree shows that M_0 is not reachable from

M in the inverse net. Thus, by Theorem 5-1, M is not reachable from M_0 in the original net.

In [CARS84], Carson has shown that reachability is decidable for a point in a geometric model of a semaphore program by working "backwards" from that point to the origin. Given a point in a geometric model of a semaphore program, we can decide the current value of the program counter in each process at that point, the value of all semaphores at that point, and the number of cycles completed by each process at that point. Since this is precisely the information contained in a marking of a bounded inverse Petri net model of a semaphore program, it is not surprising that reachability is decidable for bounded inverse Petri net models of semaphore programs using a similar method.

There exist bounded inverse Petri nets that are not models of any semaphore program. Our contribution here is to is to show how a technique similar to Carson's can be applied to a subclass of Petri nets that properly contains the subclass of bounded inverse Petri net models of semaphore programs; namely the bounded inverse subclass of Petri nets.

5.5. Chapter Summary

We have presented a new subclass of Petri nets called bounded inverse Petri nets for which reachability is decidable by using the reachability tree of [KARP69]. Our method is inspired by Carson's method for deciding the reachability of points in a geometric models of semaphore programs. We have shown that a technique similar to Carson's can be applied to the larger subclass of systems represented by bounded inverse Petri nets.

CHAPTER 6

Liveness Results for a Subclass of Petri Nets

In the last chapter we demonstrated that reachability can be decided for the dynamic subclass of bounded inverse Petri nets by an enumeration of a finite number of markings. In this chapter, we continue our study of subclasses of Petri nets. We derive necessary and sufficient conditions for the liveness of markings of a static subclass of Petri nets called *SM* (**S**ingle input **M**ultiple output) nets. Then we use these results to add *SM* nets to the list of Petri net subclasses for which the *deadlock–trap* property of [HACK72] is a necessary and sufficient condition for liveness. We use this result to show that *SM* nets can be used to analyze a subclass of semaphore programs for deadlock-freedom.

6.1. A Necessary Condition for Liveness

In this section we show that a marking of an *SM* net N is live only if there are no conflict circuits in N. Recall that a place p is a conflict place if p has more than one output transition. A circuit c is a conflict circuit if there is some conflict place $p \in c$.

Definition 6-1: SM Net

A Petri net $N = (\Pi,\Sigma,F,B)$ is an *SM* net if for all $p \in \Pi$ and for all $t \in \Sigma$, $|{\bullet}p| = 1$ and $|p{\bullet}|, |{\bullet}t|, |t{\bullet}| \geq 1$.

□

Informally, we say that a net is an *SM* net if every place has a single input transition and at least one output transition, and every transition has at least one input and output place.

The subclass of *SM* nets is properly contained in the class of backward conflict-free nets (Definition 5-9). Sufficient conditions for live and bounded markings of backward conflict-free nets have been obtained in [LIEN76]. Our work is concerned with necessary and sufficient conditions for live but not necessarily bounded markings of a sublcass of backward conflict-free nets.

We start by deriving some simple properties of *SM* nets. Our analysis centers on the circuits of the graph representation of an *SM* net. All circuits discussed in this chapter are simple circuits(defined in Section 2.4.1).

Lemma 6-1: A transition in a circuit of an *SM* net has exactly one output place in the circuit.

Proof: Clearly, for a circuit c, a transition $t \in c$ has at least one output place in c. Now suppose that t has another output place $p' \neq p$ such that $p' \in c$. Since c is simple, there exists a path from p to p' not passing through t. This implies that p' has more than one input transition. Contradiction.

□

Another property of circuits in *SM* nets is that their token counts never increase as a result of firing a transition.

Lemma 6-2: The token count of a circuit in an *SM* net cannot increase as a result of firing a transition.

Proof: Let c be a circuit and let t be a transition. If $t \notin c$, then by the definition of SM programs there is no $p \in t\bullet$ such that $p \in c$. Thus firing t cannot increase the token count of c. If $t \in c$, then by Lemma 6-1 t

has exactly one output place in c. Thus, firing t removes at least one token from c while adding exactly one token to c. Again, firing t cannot increase the token count of c.

□

We can also derive the conditions under which the token count of a circuit is guaranteed to decrease.

Lemma 6-3: Let p be a conflict place in a circuit c. Let $t \in p\bullet$ be the immediate successor of p in c and let $t' \neq t$ be a transition such that $t' \in p\bullet$. Then firing t' reduces the token count of c.

Proof: case 1: Every place in c has an input transition in c, and by the definition of *SM* nets, this transition is that place's only input transition. This implies that if $t' \notin c$, then there exists no $p' \in t'\bullet$ such that $p' \in c$. Thus, firing t' removes at least one token from c while adding no tokens to c. This results in a net loss of tokens from c.

case 2: If $t' \in c$, then there is a place $p' \neq p$ such that $p' \in c$ and $p' \in \bullet t$. By Lemma 6-1, t' has exactly one output place in c. Thus firing t' removes at least two tokens from c while adding exactly one token. Again we have a net loss of tokens from c.

□

Lemmas 6-2 and 6-3 suggest a necessary condition for the liveness *SM* net markings.

Theorem 6-1: A marking M of an *SM* net is live only if every circuit is conflict-free.

Proof: We show the contrapositive. Let p be a conflict place in a circuit c. Let $t \in p\bullet$ be the immediate successor of p in c and let $t' \neq t$ be a transition such that $t' \in p\bullet$. Now assume that M is a live marking. This implies that t' can be fired an infinite number of times from M. But by Lemmas 6-2 and 6-3, this implies that the token count of c goes to zero at some marking M' reachable from from M. By Lemma 6-2 this implies

that every transition in c is dead in M', which contradicts the assumption that M is live.

□

Recall the definition of normal Petri nets (Definition 5-14), first discussed by Yamasaki in [YAMA84]. A circuit c is minimal if the set of places in c does not properly contain the set of places of any other circuit. A Petri net is a normal net if every transition that has an input place in a minimal circuit c has an output place in c.

If every circuit in a Petri net N is conflict-free, then every minimal circuit in N is conflict-free, and N is a normal net. Now, by Theorem 6-1, a marking M of an *SM* net N is live only if every circuit of N is conflict-free. This implies that an *SM* net with a live marking is a normal net, and we can restate Theorem 6-1: A marking M of an *SM* net N is live only if N is a normal *SM* net.

6.2. Necessary and Sufficient Conditions for Liveness

We have seen that the only live markings of *SM* nets are the markings of normal *SM* nets. In this section we derive a necessary and sufficient condition for live markings of normal *SM* nets. Then we use this result and the necessary condition from the previous section to derive a necessary and sufficient condition for live markings of *SM* nets.

To derive a necessary and sufficient condition for live markings of normal *SM* nets, we show that for any normal *SM* net $N = (\Pi,\Sigma,F,B)$, there exists a sequence $w \in \Sigma^*$, such that $\overline{w} \geqslant 0$ and such that each transition in Σ occurs at least once. Using this result and a recent result from [YAMA84], we show that the absence of

tokenless circuits in a marking of a normal *SM* net is a necessary and sufficient condition for liveness.

For a normal *SM* net $N = (\Pi,\Sigma,F,B)$ and a place p, let $P(p)$ denote the set of predecessor nodes $q \neq p$ such that there is a path from q to place p. Let $S(p)$ denote the set of successor nodes $q \neq p$ such that there is a path from p to q.

First we observe that for a conflict place p, $P(p)$ and $S(p)$ are non-empty disjoint sets.

Lemma 6-4: If p is a conflict place in a normal *SM* net N, then $P(p)$ and $S(p)$ are non-empty and disjoint.

Proof: $P(p)$ and $S(p)$ are non-empty because by the definition of *SM* programs p has one input transition and at least one output transition. To see that the sets are disjoint, suppose that the contrary is true. Then there is some node $q \neq p$ such that there exists a path from q to p and a path from p to q. This implies a conflict circuit. Contradiction.

□

Next, we define some useful sets of transition sequences.

Definition 6-2: The Set of Sequences ω

For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, let ω be the infinite set of all transition sequences $w \in \Sigma^*$ such that

$$\Psi(w)(\bullet p) \geq \Psi(w)(p\bullet), \quad \forall \text{ conflict-free places } p.$$

□

In words, a sequence $w \in \omega$ is a sequence such that for all conflict-free places p, the number of instances in w of the input transition $\bullet p$ is greater than or equal to the number of instances in w of the output transition $p\bullet$. Thus for $w \in \omega$ and

conflict-free place p, $\overline{w}(p) \geqslant 0$.

Definition 6-3: The Set of Sequences $\nu(w,p)$

For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, a sequence $w \in \omega$, and a conflict place p,

$$\nu(w,p) = \{\lambda\}$$

if $\overline{w}(p) \geqslant 0$. If $\overline{w}(p) < 0$, then $\nu(w,p)$ is the finite nonempty set of sequences $v \in \Sigma^*$ such that

$$\Psi(v)(t) = \Psi(w)(p\bullet) - \Psi(w)(\bullet p), \quad \forall \text{ transitions } t \in P(p)$$

and

$$\Psi(v)(t) = 0, \quad \forall \text{ transitions } t \notin P(p)$$

□

In words, if $w \in \omega$ and p is a conflict place such that $\overline{w}(p) < 0$, then $v \in \nu(w,p)$ is a sequence containing exactly $-\overline{w}(p)$ instances of each transition in $P(p)$ and zero instances of all other transitions. If $\overline{w}(p) \geqslant 0$, then $v \in \nu(w,p)$ is simply the empty transition sequence.

We will show that for any normal *SM* net there exists a sequence w constructed by concatenating sequences from ω and ν such that $\overline{w} \geqslant 0$ and each transition in the net occurs at least once in w.

First we establish a property of sequences in ω and ν with respect to the conflict-free places of a normal *SM* net.

Lemma 6-5: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, conflict place p, $w \in \omega$, and $v \in \nu(w,p)$, $\overline{wv}(p') \geqslant 0$ for all conflict-free places p'.

Proof: For conflict-free place p', let $t = \bullet p'$ and $t' = p'\bullet$.

case 1: Suppose $\overline{w}(p) \geqslant 0$. Then by Definition 6-3 $\overline{v}(p') = 0$.

case 2: Suppose $\overline{w}(p) < 0$. By definition we have $\overline{v}(p') = \Psi(v)(t) - \Psi(v)(t')$. If $p' \in P(p)$, then $t,t' \in P(p)$, and by Definition 6-3,

$$\Psi(v)(t) = \Psi(v)(t') = \Psi(w)(p\bullet) - \Psi(w)(\bullet p)$$

which implies $\overline{v}(p') \geqslant 0$. If $p' \in S(p)$, then $t,t' \in S(p)$, and by Definition 6-3 and Lemma 6-4 $\Psi(v)(t) = \Psi(v)(t') = 0$, which implies $\overline{v}(p') = 0$. Finally, if $p' \in \neg(P(p) \cup S(p))$, then $t' \notin P(p)$, and by Definition 6-3, $\Psi(v)(t') = 0$, which implies $\overline{v}(p') = \Psi(v)(t) \geqslant 0$. In any case $\overline{v}(p') \geqslant 0$ and by Definition 6-2 $\overline{w}(p') \geqslant 0$, which implies $\overline{wv}(p') \geqslant 0$.

□

Next we establish some properties of sequences in ω and ν with respect to the conflict places of a normal *SM* net.

Lemma 6-6: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, conflict place p, $w \in \omega$, and $v \in \nu(w,p)$, $\overline{wv}(p) \geqslant \overline{w}(p)$.

Proof: case 1: Suppose $\overline{w}(p) \geqslant 0$. Then by Definition 6-3, $\overline{v}(p) = 0$, which implies $\overline{wv}(p) = \overline{w}(p)$.

case 2: suppose $\overline{w}(p) < 0$. By definition we have

$\overline{wv}(p) = \overline{w}(p) + \overline{v}(p)$

$= \Psi(w)(\bullet p) - \Psi(w)(p\bullet) + \Psi(v)(\bullet p) - \Psi(v)(p\bullet)$.

Clearly, $t \in P(p)$ for $t = \bullet p$ and $t' \notin P(p)$ for all $t' \in p\bullet$. Thus, by Definition 6-3 we have

$\Psi(v)(\bullet p) = \Psi(w)(p\bullet) - \Psi(w)(\bullet p)$

and

$\Psi(v)(p\bullet) = 0$

which implies

$\overline{wv}(p) = \Psi(w)(\bullet p) - \Psi(w)(p\bullet) + \Psi(w)(p\bullet) - \Psi(w)(\bullet p) - 0 = 0 > \overline{w}(p)$.

In either case $\overline{wv}(p) \geqslant \overline{w}(p)$.

□

Lemma 6-7: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, conflict place $p \in \Pi$, $w \in \omega$, and $v \in \nu(w,p)$, $\overline{wv}(p') = \overline{w}(p')$ for all conflict places $p' \in S(p)$.

Proof: case 1: If $\overline{w}(p) \geqslant 0$, then by Definition 6-3, $\overline{v}(p') = 0$.

case 2: If $\overline{w}(p) < 0$, then by Definition 6-3 and Lemma 6-4, $\overline{v}(p') = 0$.

In either case, $\overline{wv}(p') = \overline{w}(p')$.

□

Lemma 6-8: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, conflict place $p \in \Pi$, $w \in \omega$, and $v \in \nu(w,p)$, $\overline{wv}(p') \geqslant \overline{w}(p')$ for all conflict places $p' \in \neg(P(p) \cup S(p))$.

Proof: case 1: Suppose $\overline{w}(p) \geqslant 0$. Then by Definition 6-3, $\overline{v}(p') = 0$.

case 2: Suppose $\overline{w}(p) < 0$. By assumption there is no $t' \in p'\bullet$ such that $t' \in P(p)$. By definition

$\overline{v}(p') = \Psi(v)(\bullet p') - \Psi(v)(p'\bullet)$

which by Definition 6-3 implies

$= \Psi(v)(\bullet p') - 0 \geqslant 0.$

In either case, $\overline{wv}(p') \geqslant \overline{w}(p')$.

□

Next we introduce the notions of a minimal conflict place and the level of a conflict place.

Definition 6-4: Minimal Conflict Place

A conflict place $p \in \Pi$ in a normal *SM* net is a minimal conflict place if there exist no conflict places in $S(p)$.

□

Notice that the absence of conflict circuits in normal *SM* nets implies that a normal *SM* net contains conflict places if and only if it contains at least one minimal conflict place.

Definition 6-5: Level of a Conflict Place

For a conflict place p in a normal *SM* net, let $H(p)$ be the set of all simple paths of the form $p \cdots p'$, where p' is a minimal conflict place. For some $h \in H(p)$, let $n(h)$ denote the number of conflict places in h minus unity. Then the level of a conflict place p is denoted level(p) and defined by

$$\text{level}(p) = \begin{cases} 0 & \text{if } p \text{ is minimal} \\ \max(n(h), h \in H(p)) & \text{otherwise} \end{cases}$$

□

Notice that the absence of conflict circuits in normal *SM* nets implies that if p is a conflict place, then p is either minimal or there is a minimal conflict place in $S(p)$. Thus level(p) is defined for all conflict places p in a normal *SM* net.

Our next result establishes properties of sequences in ω and ν with respect to conflict places and their levels.

Lemma 6-9: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, a conflict place $p \in \Pi$, $w \in \omega$, and $v \in \nu(w,p)$, if level(p') $\leqslant$ level(p), then $\overline{wv}(p') \geqslant \overline{w}(p')$ for all conflict places p'.

Proof: Suppose the contrary, such that $\overline{wv}(p') < \overline{w}(p')$. By Lemma 6-6,

$p' \neq p$. By Lemma 6-7, $p' \notin S(p)$. By Lemma 6-8, $p' \notin \neg(P(p) \cup S(p))$. Thus $p' \in P(p)$. But this implies a path from p' to p, which implies level(p') > level(p). Contradiction.

□

Finally, we show that for any normal *SM* net we can always construct a sequence w containing at least one instance of each transition of the net such that $\overline{w} \geqslant 0$.

Lemma 6-10: For a normal *SM* net $N = (\Pi,\Sigma,F,B)$, there exists a sequence of transitions $w \in \Sigma^*$ such that $\Psi(w)(t) \geqslant 1$ for all $t \in \Sigma$, and $\overline{w} \geqslant 0$.

Proof: Let $cp^i = p_1 \cdots p_i$ be any sequence of conflict places such that level(p_j) $\leqslant$ level(p_k), for all $j \leqslant k \leqslant i$. Let w^0 be a sequence such that $\Psi(w^0)(t) = 1$ for all $t \in \Sigma$. For $i > 0$, let $w^i = w^{i-1}v_i$, where $v_i \in \nu(w^{i-1},p_i)$. We claim that for any normal *SM* net containing at least n conflict places, there exist cp^n and w^n such that $w^n \in \omega$ and $\overline{w}^n(p_i) \geqslant 0$ for all conflict places $p_i \in cp^n$.

Clearly cp^n exists, since there are at least n conflict places and since level(p) is defined for each conflict place p. Next we use induction on n to prove our assertion about w^n.

Basis: Net N contains at least zero conflict places. Clearly w^0 exists. Now, by the definition of w^0, $\Psi(w^0)(t) = 1$ for all $t \in \Sigma$. For conflict-free place p, this implies $\Psi(w^0)(\bullet p) = \Psi(w^0)(p\bullet)$, which implies $\overline{w}^0(p) = 0$, which implies $w^0 \in \omega$. Since $cp^0 = \lambda$, $\overline{w}^0(p_i) \geqslant 0$ for all conflict places $p_i \in cp^0$.

Induction Hypothesis: Net N contains at least k conflict places. Assume that there exists a w^k such that $w^k \in \omega$ and $\overline{w}^k(p_i) \geqslant 0$, for all conflict places $p_i \in cp^k$.

Induction Step: Net N contains at least $k + 1$ conflict places. Clearly w^{k+1} exists. By the induction hypothesis there exists some $w^k \in \omega$, which by Lemma 6-5 implies $\overline{w}^{k+1}(p) \geqslant 0$ for all conflict-free places p which implies $w^{k+1} \in \omega$. Now level(p_i) $\leqslant$ level(p_{k+1}) for all $p_i \in cp^{k+1}$ by the definition of cp^{k+1}, and $w^k \in \omega$ by the induction hypothesis. Then by Lemma 6-9 and the induction hypothesis, $\overline{w}^{k+1}(p_i) \geqslant \overline{w}^k(p_i)$ for all $p_i \in cp^{k+1}$.

Thus, for a normal *SM* net with n conflict places, a sequence w^n exists

such that $w^n \in \omega$. By Definition 6-2 this implies $\overline{w}^n(p) \geqslant 0$ for all conflict-free places $p \in \Pi$. Further, by the induction proof, $\overline{w}^n(p_i) \geqslant 0$ for all conflict places $p_i \in cp^n$. Thus, $\overline{w}^n \geqslant 0$. Trivially, $\Psi(w^k)(t) \geqslant 1$ for all $t \in \Sigma$, by the definition of w^0.

□

In [YAMA84], Yamasaki shows that if every circuit of a normal Petri net N is marked in a marking M, then a rearrangement of a sequence w is enabled in M if and only if $M + \overline{w} \geqslant 0$. We use Yamasaki's recent result, with Lemma 6-2 and Lemma 6-10, to establish a necessary and sufficient condition for a live marking of a normal *SM* net.

Theorem 6-2: A marking M of a normal *SM* net N is live if and only if every circuit is marked in M.

Proof:

If: Assume that every circuit of a normal *SM* net is marked and assume that M is not a live marking. Then there exists a marking M' reachable from M such that some transition t is dead in M'. By Lemma 6-10, there exists a sequence $w \in \Sigma^*$ such that $t \in w$ and $\overline{w} \geqslant 0$, which implies that $M' + \overline{w} \geqslant 0$. By Yamasaki's result, there exists a rearrangement of $\overline{w}$ which is enabled in M'. But this implies that t is not dead in M'. Contradiction.

Only If: Assume that M is a live marking. Now suppose that there exists a circuit c unmarked in M. By Lemma 6-2 this implies that every transition in c is dead. This contradicts the assumption that M is a live marking.

□

We use to Theorems 6-1 and 6-2 to prove the central result of this section.

Theorem 6-3: A marking M of an *SM* net N is live if and only if there are no conflict circuits in N and all circuits are marked in M.

Proof:

If: See Theorem 6-2.

Only If: We prove the contrapositive. If there is a conflict circuit in N, then by Theorem 6-1, M is not live. If there are no conflict circuits in N, and there is an circuit unmarked in M, then by Theorem 6-2, M is not live.

□

For a simple application of Theorem 6-3, consider the *SM* nets shown in Figure 6-1. The marking of the net in Figure 6-1(a) is not live because the circuit $p_1 t_1 p_2 t_2 p_1$ contains the conflict place p_2. The marking of the net in Figure 6-1(b) is not live because the circuit $p_1 t_1 p_3 t_2 p_1$ is tokenless. The marking of the net in Figure 6-1(c) is live because every circuit is conflict-free and every circuit contains at least one token. Unfortunately the liveness conditions in Theorem 6-3 do not hold for normal nets in general.

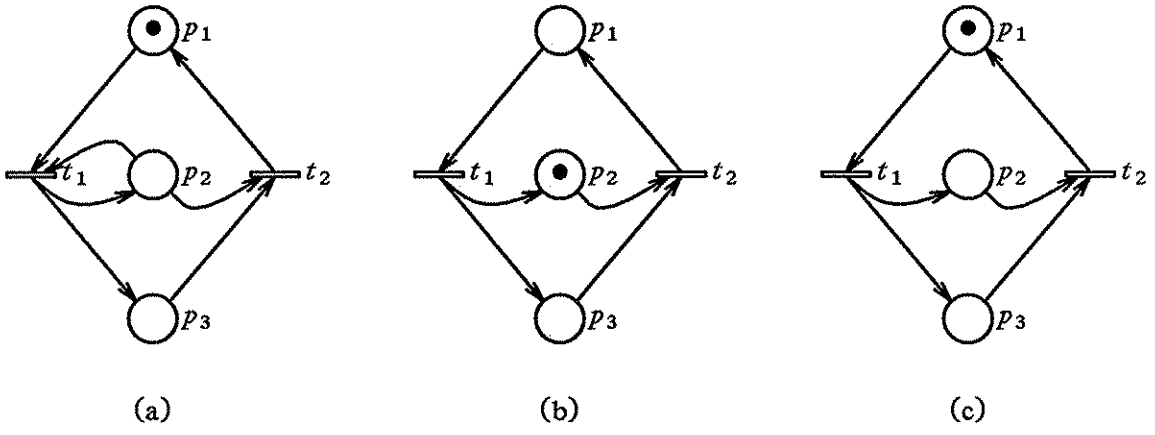


Figure 6-1: *Examples of SM Nets.*

6.3. SM Nets and the Deadlock-Trap Property

The *deadlock–trap* property of Petri net markings was first described by Hack in [HACK72], where it was shown to be a necessary and sufficient condition for live markings of free-choice nets(Definition 5-10). It is also known that the deadlock-trap property is a necessary and sufficient condition for live markings of CNI nets(Definition 5-5), and NSK nets(Definition 5-11), and a sufficient condition for live markings of extended simple nets(Definition 5-13). The unified sufficiency proof[JANT79] relies on a property of these nets that does not hold for *SM* nets. However, we are able to use Theorem 6-3 to add *SM* nets to the subclasses of Petri nets for which the deadlock-trap property is a necessary and sufficient condition for liveness. First, we describe the deadlock-trap property.

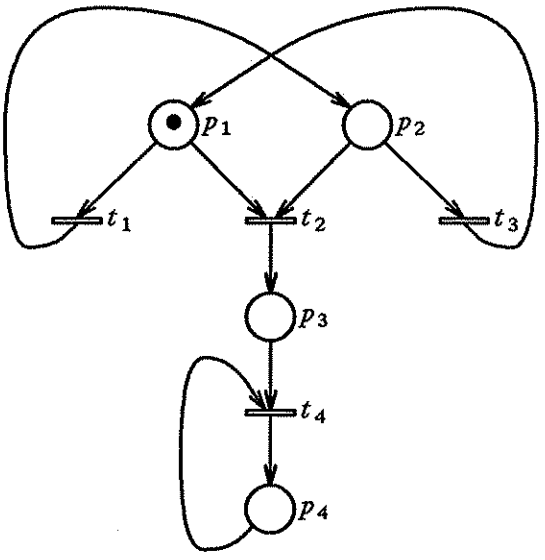


Figure 6-2: *A Counter-Example.*

For a Petri net N, a *deadlock*‡ is a nonempty set of places A such that $\bullet A \subseteq A\bullet$. A *trap* is a nonempty set of places A such that $A\bullet \subseteq \bullet A$. A trap A is marked in marking M if at least one of the places in A is marked. A marking M of net N satisfies the deadlock-trap property if and only if every deadlock contains a marked trap.

The unified sufficiency proof in [JANT79] relies on the following property of maximal markings(Definition 4-3) of free-choice nets, NSK nets, CNI nets, and extended simple nets:

P1: For a free-choice, NSK, CNI, or extended simple Petri net N and a maximal marking M (Definition 4-3), for all transitions t dead in M there exists some input place $p \in \bullet t$ such that p is unmarked in all markings reachable from M'.

The *SM* net in Figure 6-2 is a simple counter-example showing that this property does not hold for *SM* nets. From the marking $M = (1,0,0,0)$, there are only two reachable markings: M and $M' = (0,1,0,0)$. Transitions t_1 and t_3 are live in M and M'. Transitions t_2 and t_3 are dead in M and M'. Thus M is a maximal marking. Now t_2 is dead in M, yet neither of its input places is unmarked in both M and M'. Thus property $P1$ does not hold for *SM* nets and the approach in [JANT79] is not appropriate for *SM* programs.

Fortunately, we can use Theorem 6-3 and some simple static properties of *SM* nets to show that the deadlock-trap property is a necessary and sufficient condition for live markings of *SM* nets. First, we establish a relationship between circuits

‡ The choice of terminology here is unfortunate, but it has become standard in the Petri net literature.

and minimal conflict circuits.

For a circuit c, let C denote the set of all places in c.

Lemma 6-11: If a circuit c in an *SM* net $N = (\Pi,\Sigma,F,B)$ is not minimal, then there exists a minimal conflict circuit c_m such that C_m is a proper subset of C.

Proof: If c is not minimal, then there is some circuit c' such that C' is a proper subset of C. If c' is not minimal, then there is some circuit c'' such that C'' is a proper subset of C', and so on. Eventually we get a minimal circuit c_m such that C_m is a proper subset of C. This implies that for $p \in c_m$, and $p' \in c$, $p' \notin c_m$, there is a path from p to p'. Now suppose that c_m is conflict-free. This implies that a path from p to p' is of the form $p \cdots t\, p'' \cdots p'$, where $t \in c_m$, and $p'' \in c$, $p'' \notin c_m$. In addition to p'', t also has an output place in c_m, which implies that t has at least two output places in c, which by Lemma 6-1 implies that N is not an *SM* net. Contradiction.

□

Next we establish the conditions under which the set of places in a minimal circuit contains a trap.

Lemma 6-12: The set of places C of a minimal circuit c in an *SM* net contains a trap if and only if c is conflict-free.

Proof:

If: Suppose that c is a minimal conflict-free circuit. Clearly, $C\bullet = \bullet C$, which implies that C is a trap.

Only If: Suppose that for a minimal circuit c, C contains a trap A such that $A\bullet \subseteq \bullet A$. Since c is minimal, a proper subset of the places in c can not be a trap, so $A = C$. Now suppose that there is a conflict place $p \in c$. Let $t \in p\bullet$ be the successor of p in c and consider some $t' \neq t$ such that $t' \in p\bullet$. If $t' \in c$, then c is not minimal. Contradiction. If $t' \notin c$, then $t' \notin \bullet A$, by the definition of *SM* nets. This implies $A\bullet \not\subseteq \bullet A$. Contradiction.

□

Next, we establish a rather obvious relationship between circuits and deadlocks.

Lemma 6-13: If c is a circuit, then C is a deadlock.

Proof: Let c be a circuit. Now suppose that C is not a deadlock. That is, $\bullet C \not\subseteq C\bullet$. Then there exists a $p \in C$ with $t = \bullet p \in \bullet C$ and $t \notin C\bullet$. Since t is the only input transition of p, then for all $p' \neq p \in C$ there is no path from p' to p, which contradicts the assumption that c is a circuit.

□

We use Lemmas 6-11, 6-12, and 6-13, along with Theorem 6-3, to show that the deadlock trap property is a necessary and sufficient liveness condition for *SM* programs.

Theorem 6-4: A marking M of an *SM* net N is live if and only if M satisfies the deadlock-trap property.

Proof:

If: We show the contrapositive. Suppose that M is not a live marking. Then by Theorem 6-3 there is a conflict circuit in N or a circuit unmarked in M.

case 1: Suppose that there is a conflict circuit c in N. By Lemma 6-11 there exists a minimal conflict circuit c_m. By Lemma 6-12, C_m contains no traps. By Lemma 6-13, C_m is a deadlock. Thus M does not satisfy the deadlock-trap property.

case 2: Suppose that there is a circuit c unmarked in M. By Lemma 6-13, C is a deadlock. Since all places in C are unmarked in M, C cannot contain a marked trap. Thus M does not satisfy the deadlock-trap property.

Only If: Suppose that M is a live marking. Then by Theorem 6-3 there are no conflict circuits in N and every circuit is marked in M. Now suppose that M does not satisfy the deadlock-trap property. Then there exists a

deadlock A, $\bullet A \subseteq A\bullet$, that contains no marked traps. Let $p \in A$ with $t = \bullet p \in \bullet A$. Since A is a deadlock, there is some $p' \in A$ such that $t \in p'\bullet$, and $t' = \bullet p' \in \bullet A$. Since A is a deadlock, there is some $p'' \in A$ such that $t' \in p''\bullet$ and $t'' = \bullet p'' \in A$, and so on. Eventually we get a circuit c. By the contrapositive of Lemma 6-11, c is minimal. By Lemma 6-12, C is a trap, and by assumption, C is a marked trap. Contradiction.

□

For some simple applications of Theorem 6-4, consider the *SM* nets shown in Figure 6-1. For the net in Figure 6-1(a) with initial marking $M = (1,0,0)$, let $A = \{p_2\}$ with $\bullet A = \{t_1\}$ and $A\bullet = \{t_1,t_2\}$. We see that A is a deadlock containing no traps, and conclude that M is not live. For the net in Figure 6-1(b) with initial marking $M = (0,1,0)$, let $A = \{p_1,p_4\}$ with $\bullet A = A\bullet = \{t_1,t_2\}$. Now A is a deadlock and a trap, yet A is not a marked trap and neither $\{p_1\}$ nor $\{p_3\}$ is a trap. Thus A is a deadlock containing no marked traps, and we conclude that M is not live. Finally, consider the net in Figure 6-1(c) with initial marking (1,0,0). The analysis for satisfaction of the deadlock-trap property is given below.

Analysis for the Deadlock-Trap Property

A	$\bullet A$	$A\bullet$	Deadlock?	Marked Trap?
$\{p_1\}$	$\{t_2\}$	$\{t_1\}$	no	
$\{p_2\}$	$\{t_1\}$	$\{t_2\}$	no	
$\{p_3\}$	$\{t_1\}$	$\{t_2\}$	no	
$\{p_1,p_2\}$	$\{t_1,t_2\}$	$\{t_1,t_2\}$	yes	yes
$\{p_2,p_3\}$	$\{t_1\}$	$\{t_2\}$	no	
$\{p_1,p_3\}$	$\{t_1,t_2\}$	$\{t_1,t_2\}$	yes	yes
$\{p_1,p_2,p_3\}$	$\{t_1,t_2\}$	$\{t_1,t_2\}$	yes	yes

Since every deadlock contains a marked trap, we conclude that M is live.

6.4. SM Nets and Semaphore Programs

In this section we discuss how *SM* nets can be used to model a subclass of general resource semaphore programs called *SM* (**S**ingle release **M**ultiple requests) programs.

Definition 6-6: SM Programs

A semaphore program C is an *SM* program if for each semaphore σ of C, exactly one statement releases σ and at least one statement requests σ.

□

Notice that the subclass of *SM* programs properly includes the subclass of *SS* programs (Definition 4-1).

We argue that the Petri net model of an *SM* program is an *SM* net (Definition 6-1). By the construction of Petri net models of semaphore programs outlined in Section 2.4.2, we note that each program counter place has exactly one input transition and exactly one output transition, regardless of the semaphore program being modeled. Also by the construction, we see that if k statements release semaphore σ, then the corresponding semaphore place in the Petri net model has exactly k input transitions. Similarly, if k statements request semaphore σ, then the corresponding semaphore place in the Petri net model has exactly k output transitions. Since each semaphore in an *SM* program is released by exactly one statement and requested by at least one statement, we see that every semaphore place in the Petri net model of an *SM* program has exactly one input transition and at least one output transition. Hence, all places in the Petri net model of an *SM* program have exactly one input transition and at least one output transition, which satisfies the definition of an *SM* net.

```
e = 0, a = 1, b = 0: semaphore
cobegin                         — SM Program
        cycle                   — producer process
                V(e)            — produce an element and place it in the buffer
        endcycle
        //
        cycle                   — consumer process #1
                P(a)            — wait for this consumer's turn to come around
                P(e)            — remove an element from the buffer and consume it
                V(b)            — allow the next consumer process to proceed
        endcycle
        //
        cycle                   — consumer process #2
                P(b)            — wait for this consumer's turn to come around
                P(e)            — remove an element from the buffer and consume it
                V(a)            — allow the next consumer process to proceed
        endcycle
coend
```

Figure 6-3: *A Correct SM Program.*

```
e = 0, a = 0, b = 0: semaphore
cobegin                          — SM Program
        cycle                    — producer process
                V(e)             — produce an element and place it in the buffer
        endcycle
        //
        cycle                    — consumer process #1
                P(a)             — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V(b)             — allow the next consumer process to proceed
        endcycle
        //
        cycle                    — consumer process #2
                P(b)             — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V(a)             — allow the next consumer process to proceed
        endcycle
coend
```

Figure 6-4: *An Incorrect SM Program.*

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V(e)	t_1	P(a)	t_4	P(b)	P_8	e
		t_2	P(e)	t_5	P(e)	P_9	a
		t_3	V(b)	t_6	V(a)	P_{10}	b

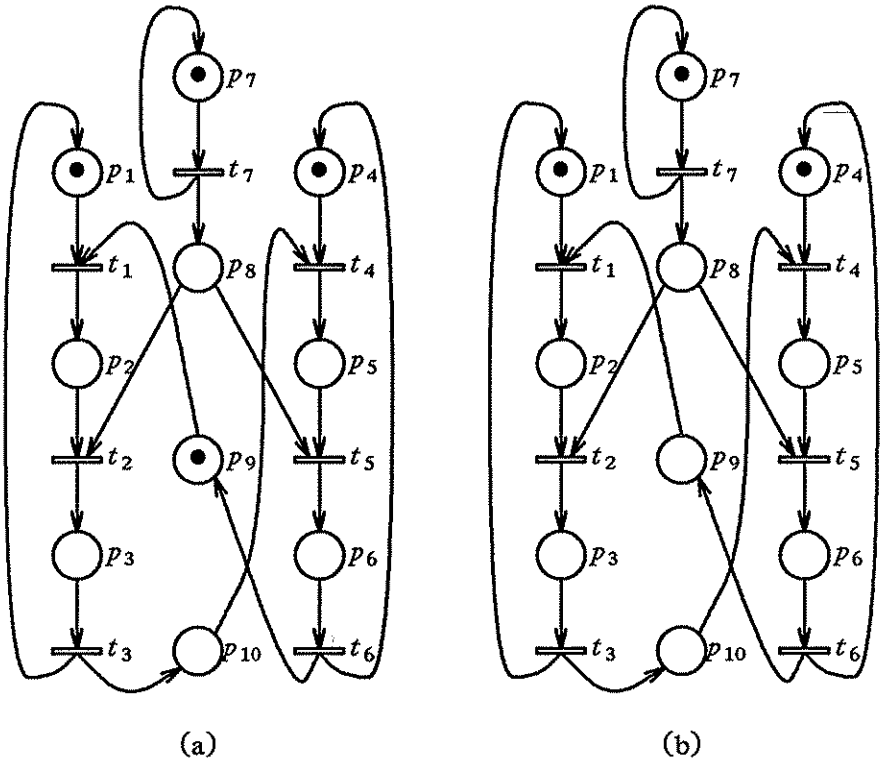


Figure 6-5: *Petri Net Models of Correct and Incorrect SM Programs.*

Consider a three-process producer/consumer system where a producer process deposits items into an infinite buffer and where two consumer processes take turns removing items from the buffer. The synchronizing behavior of such a system is captured by the *SM* program in Figure 6-3.

To test this *SM* program for deadlock potential, we construct the *SM* Petri net model with initial marking M_0 shown in Figure 6-5(a). Since every circuit is conflict-free and marked in M_0 we conclude that M_0 is live and that the program is deadlock-free.

Now suppose that we neglect to initialize semaphore b, as shown in Figure 6-4. As before we construct the *SM* Petri net model with initial marking M_0 as shown in Figure 6-5(b). Since there is a circuit unmarked in M_0, we conclude that M_0 is not live and that the *SM* program is not deadlock free.

6.5. Chapter Summary

We have identified a subclass of Petri nets called *SM* nets and have derived necessary and sufficient conditions for the liveness of their markings. We used these results to add *SM* nets to the list of Petri net subclasses for which the deadlock-trap property is a necessary and sufficient condition for liveness. Finally, we used these results to demonstrate that *SM* nets can be used to model a subclass of semaphore programs for deadlock-freedom.

CHAPTER 7

Static Incremental Deadlock Detection

In this chapter, we use the results from the previous chapter to show how efficient static incremental deadlock detection might be performed on the subclass of *SM* semaphore programs.

7.1. Static Incremental Deadlock Detection

Reynolds has proposed a class of concurrent programs in [REYN79] where the number of processes is fixed and known at compile time, as are the *interconnections* among processes. An interconnection represents some form of potential synchronization among a collection of processes. For example, an interconnection in a semaphore program might be defined by a semaphore σ, its initial value σ_0, and the collection of statements that request and release σ.

Reynolds views the construction of a program as an incremental activity comprising a sequence of steps. Each step is a either a definition of a process type with unspecified interconnections, an instantiation of a previously defined process type, or the addition or removal of an interconnection to or from an instantiation of a previously defined process type.

This view of program construction is appealing because it allows the designer to maintain a library of frequently used process types, which can can then be instantiated whenever they are needed for a particular program.

Let us construct the example *SM* program of Section 6-4 using Reynolds's incremental view of program construction. Recall that the example program consists of a producer process that deposits items into an infinite buffer, and two consumer processes that take turns removing items from the buffer.

We start with a semaphore program consisting of an empty list of semaphores and an empty **cobegin** statement.

```
cobegin
coend
```

Next we define a producer process type

```
cycle
        V()     — produce an element and place it in the buffer
endcycle
```

and a consumer process type

```
cycle
        P()     — wait for this consumer's turn to come around
        P()     — remove an element from the buffer and consume it
        V()     — allow the next consumer process to proceed
endcycle
```

A statement such as $P()$ or $V()$ is akin to a *skip* statement[DIJK76], having no effect other than to advance the program counter to the next statement. Next, we create an instance of the producer process and two instances of the consumer process inside the **cobegin** statement.

```
cobegin
        cycle           — producer process
                V()     — produce an element and place it in the buffer
        endcycle
        //
        cycle           — consumer process #1
                P()     — wait for this consumer's turn to come around
                P()     — remove an element from the buffer and consume it
                V()     — allow the next consumer process to proceed
        endcycle
        //
        cycle           — consumer process #2
                P()     — wait for this consumer's turn to come around
                P()     — remove an element from the buffer and consume it
                V()     — allow the next consumer process to proceed
        endcycle
coend
```

For semaphore programs an interconnection is defined by a semaphore, its initial value, and the collection of statements that either request or release the semaphore. To add an interconnection to a semaphore program, we add a semaphore σ and its initial value σ_0 to the list of semaphores, and we indicate which statements request or release σ. To remove an interconnection from a semaphore program, we remove a semaphore σ from the list of semaphores, and we remove σ from every statement that requests or releases σ.

First we add the the interconnection associated with the buffer semaphore.

```
e = 0: semaphore
cobegin
        cycle           — producer process
                V(e)    — produce an element and place it in the buffer
        endcycle
        //
        cycle           — consumer process #1
                P()     — wait for this consumer's turn to come around
                P(e)    — remove an element from the buffer and consume it
                V()     — allow the next consumer process to proceed
        endcycle
        //
        cycle           — consumer process #2
                P()     — wait for this consumer's turn to come around
                P(e)    — remove an element from the buffer and consume it
                V()     — allow the next consumer process to proceed
        endcycle
coend
```

Next we add the interconnection associated with the semaphore that allows the first consumer process to proceed.

```
e = 0, a = 1: semaphore
cobegin
        cycle           — producer process
                V(e)    — produce an element and place it in the buffer
        endcycle
        //
        cycle           — consumer process #1
                P(a)    — wait for this consumer's turn to come around
                P(e)    — remove an element from the buffer and consume it
                V()     — allow the next consumer process to proceed
        endcycle
        //
        cycle           — consumer process #2
                P()     — wait for this consumer's turn to come around
                P(e)    — remove an element from the buffer and consume it
                V(a)    — allow the next consumer process to proceed
        endcycle
coend
```

Finally we add the interconnection associated with the semaphore that allows the second consumer process to proceed.

```
e = 0, a = 1, b = 0: semaphore
cobegin
        cycle           -- producer process
                V(e)    -- produce an element and place it in the buffer
        endcycle
        //
        cycle           -- consumer process #1
                P(a)    -- wait for this consumer's turn to come around
                P(e)    -- remove an element from the buffer and consume it
                V(b)    -- allow the next consumer process to proceed
        endcycle
        //
        cycle           -- consumer process #2
                P(b)    -- wait for this consumer's turn to come around
                P(e)    -- remove an element from the buffer and consume it
                V(a)    -- allow the next consumer process to proceed
        endcycle
coend
```

We have built the example program by incrementally adding interconnections to the instantiated process types. Each new interconnection has the potential to introduce deadlock into the program. As each new interconnection is added, we would like to verify that the program is deadlock-free. This activity is called *static incremental deadlock detection.*

The motivation behind all static modeling and analysis methods is to detect errors early in the program development cycle. The notion of static incremental deadlock detection extends this reasoning even further by detecting errors as soon as they are introduced during the construction of the program.

We know of no work in the literature where static incremental deadlock detection is performed within the context of Reynold's view of programs and their construction. In the communications literature, rules have been developed in [SIDH82], [ZAFI80], [GOUD83], and [CHOW84] for incrementally designing deadlock-free communications protocols. However, these rules are only sufficient conditions

for deadlock freedom. Further, there is no notion of process types and their instantiations. The Poker environment of [SNYD84] allows the designer to describe process interconnections through a graphics interface; however, Poker provides no facilities for incrementally detecting deadlock.

7.2. An Example of Static Incremental Deadlock Detection

In this section, we use the results of Chapter 6 to show how static incremental deadlock detection might be performed on the subclass of semaphore programs that can be modeled with *SM* nets, namely, the subclass of *SM* programs.

Recall from Theorem 6-3 that a marking M of an *SM* net is live if and only if N contains no conflict circuits and every circuit is marked in M. In practice, these conditions can be checked without enumerating all of the circuits in N.

The *unmarked subnet* of net $N = (\Pi,\Sigma,F,B)$ with marking M is denoted $U=(\Pi_U,\Sigma,F_U,B_U)$. It is the Petri net formed by removing from N all places marked in M and their input and output arcs.

If U is acyclic, then the *topological ordering* of U is a sequence of nodes $T = n_1 \cdots n_k$ such that each n_i is distinct, $k = |\Pi_U \cup \Sigma|$, and for all $i \leqslant j$, there is no path from n_j to n_i. If U is cyclic, then $T = \lambda$. The topological ordering of U can be computed in polynomial time[AHO83].

The set of *strong components* of N is denoted S. Each $S_i \in S$ is a maximal subnet of N in which there is a path from any one node in S_i to any other node in S_i. The set of strong components of N can be computed in polynomial time[AHO83].

It is easy to show that for a net N and a marking M, N contains a circuit unmarked in M if and only if the unmarked subnet U is cyclic. It is well known that a graph is acyclic if and only if it admits a topological ordering[PFAL77]. Thus a topological sort[AHO83] of the unmarked subnet U tells us if N contains any circuits that are unmarked in M.

It is also easy to show that a net N contains no conflict circuits if and only if for all conflict places p, p is the only node in its strong component.

The properties of unmarked graphs and strong components and the results of Theorem 6-3 suggest a simple polynomial-time test for the liveness of a marking M of an SM net N.

First we construct the unmarked subnet of N, and attempt to generate a topological ordering of the nodes in the unmarked subgraph. If such an ordering exists, then N contains no circuits unmarked in M. Otherwise, N contains circuits unmarked in M and by Theorem 6-3, M is not a live marking.

If N contains no circuits unmarked in M, then we check for conflict circuits. We generate the strong components of N and then look for a conflict place that is not the only node in its strong component. If there exists such a conflict place, then there exists a conflict circuit in N, and by Theorem 6-3 M is not a live marking. Otherwise, by Theorem 6-3 M is a live marking.

We use this liveness test as the basis for performing static incremental deadlock detection on SM programs.

We start with an *SM* program C^0, where all process types have been instantiated and where no interconnections have been defined. Such a program is always deadlock-free.

From C^0, we build its *SM* net model N^0 with initial marking M_0^0, the unmarked subnet U^0 and its its topological ordering T^0, and the set of strong components S^0. Since C^0 is deadlock-free, we set boolean $live^0 = \textbf{true}$ to indicate that C^0 is deadlock-free.

At the *ith* step in the construction process we have the *SM* program C^i, *SM* net N^i, M_0^i, U^i, T^i, and S^i. We can either add a new interconnection to program C^i or we can delete an existing interconnection from C^i to get the new *SM* program C^{i+1}.

Suppose we delete an existing interconnection from C^i. Then we remove the associated semaphore place p and its input and output arcs from N^i to get net N^{i+1} and the new initial marking M_0^{i+1}. We then construct U^{i+1}, T^{i+1}, and S^{i+1}, and apply the liveness test for *SM* nets to determine $live^{i+1}$. Note that $live^i$ implies $live^{i+1}$.

Suppose we add an interconnection to C^i. Then we add the associated semaphore place p and its input and output arcs to N^i to get N^{i+1} and the new initial marking M_0^{i+1}. Next we construct U^{i+1}, T^{i+1} and S^{i+1} and apply the liveness test for *SM* programs to determine $live^{i+1}$. Note that $\neg live^i$ implies $\neg live^{i+1}$.

Consider how we might perform static incremental deadlock on the example program of the previous section.

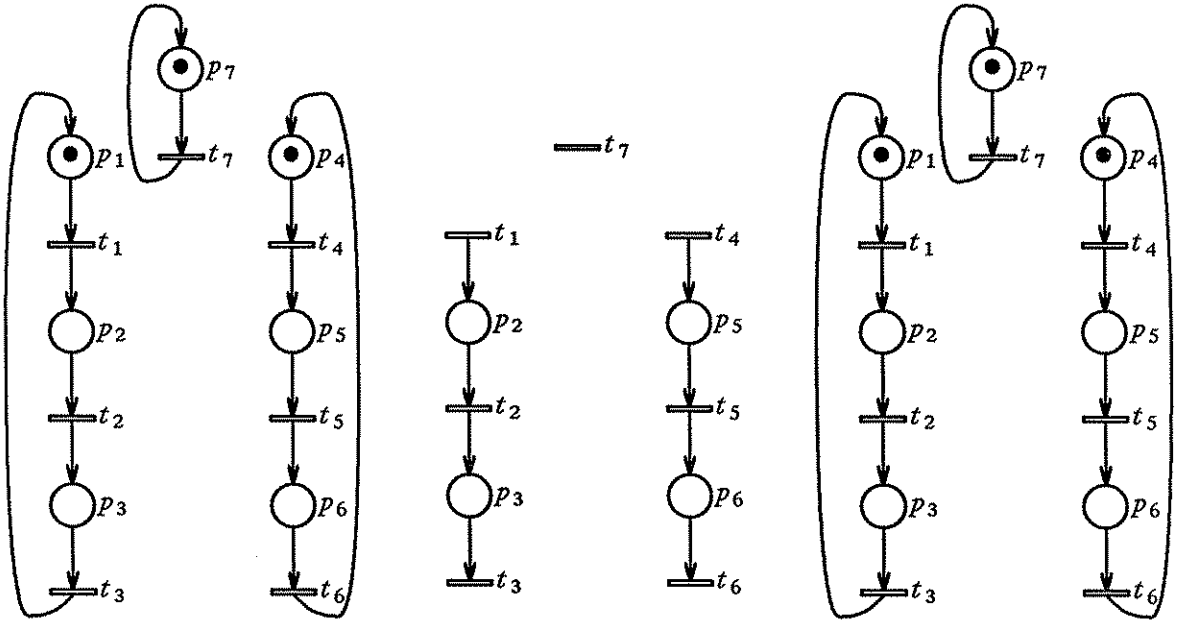
We start with the *SM* program C^0 in which all process types have been defined and instantiated and in which no interconnections have been defined.

```
cobegin                    — SM Program C^0
        cycle              — producer process
                V()        — produce an element and place it in the buffer
        endcycle
        //
        cycle              — consumer process #1
                P()        — wait for this consumer's turn to come around
                P()        — remove an element from the buffer and consume it
                V()        — allow the next consumer process to proceed
        endcycle
        //
        cycle              — consumer process #2
                P()        — wait for this consumer's turn to come around
                P()        — remove an element from the buffer and consume it
                V()        — allow the next consumer process to proceed
        endcycle
coend
```

We construct N^0, M_0^0, U^0, T^0, and S^0, as shown in Figure 7-1. No liveness test is necessary, since *SM* programs with no interconnections defined are always deadlock-free.

Next we we add the interconnection associated with the buffer semaphore.

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V()	t_1 t_2 t_3	P() P() V()	t_4 t_5 t_6	P() P() V()		



(a) *SM Net* N^0 (b) *Unmarked Subnet* U^0 (c) *Strong Components* S^0

$$T^0 = t_1 p_2 t_2 p_3 t_3 t_4 p_5 t_5 p_6 t_6 t_7$$

(d) *Topological Ordering* T^0

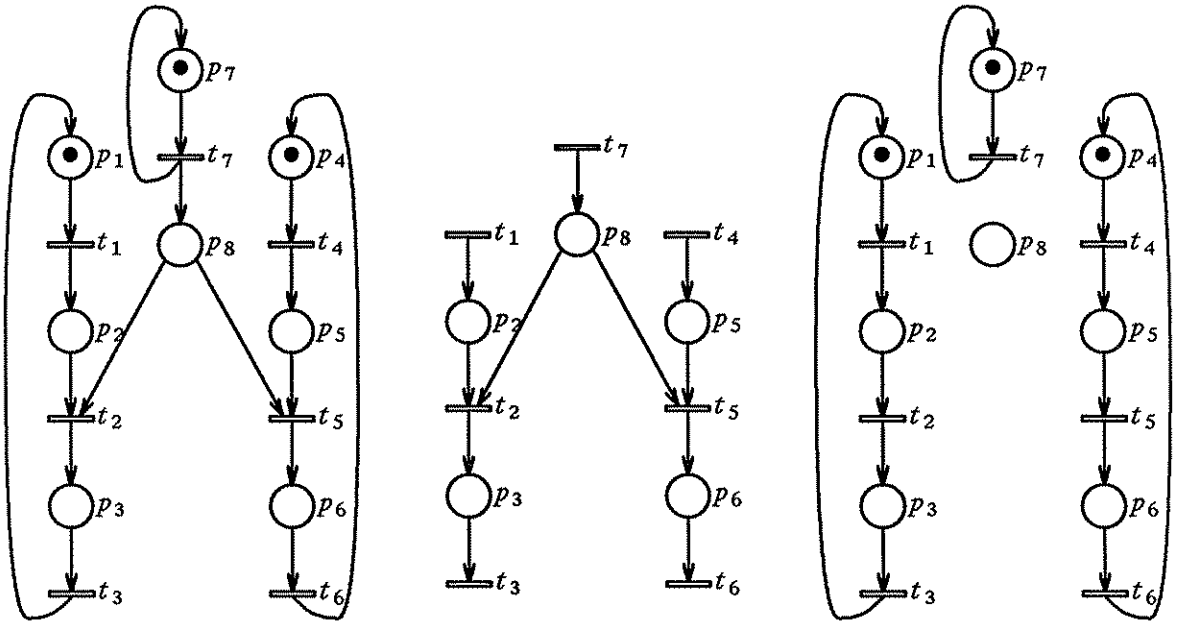
Figure 7-1: *Liveness Test For SM Program* C^0.

```
e = 0: semaphore
cobegin                          — SM Program C¹
        cycle                    — producer process
                V(e)             — produce an element and place it in the buffer
        endcycle
        //
        cycle                    — consumer process #1
                P()              — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V()              — allow the next consumer process to proceed
        endcycle
        //
        cycle                    — consumer process #2
                P()              — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V()              — allow the next consumer process to proceed
        endcycle
coend
```

We construct N^1, M_0^1, U^1, T^1, and S^1 as shown in Figure 7-2. Since $T^1 \neq \lambda$, N^1 has no circuits unmarked in M_0^1. Since conflict place p_8 is the only node in its strong component, N^1 has no conflict circuits. Hence $live^1 = \textbf{true}$ and C^1 is deadlock-free.

Next, we add the interconnection associated with the semaphore that allows the first consumer process to proceed.

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V(e)	t_1 t_2 t_3	P() P(e) V()	t_4 t_5 t_6	P() P(e) V()	p_8	e



(a) *SM Net* N^1 (b) *Unmarked Subnet* U^1 (c) *Strong Components* S^1

$$T^1 = t_1 p_2 t_4 p_5 t_7 p_8 t_2 t_5 p_3 t_3 p_6 t_6$$

(d) *Topological Ordering* T^1

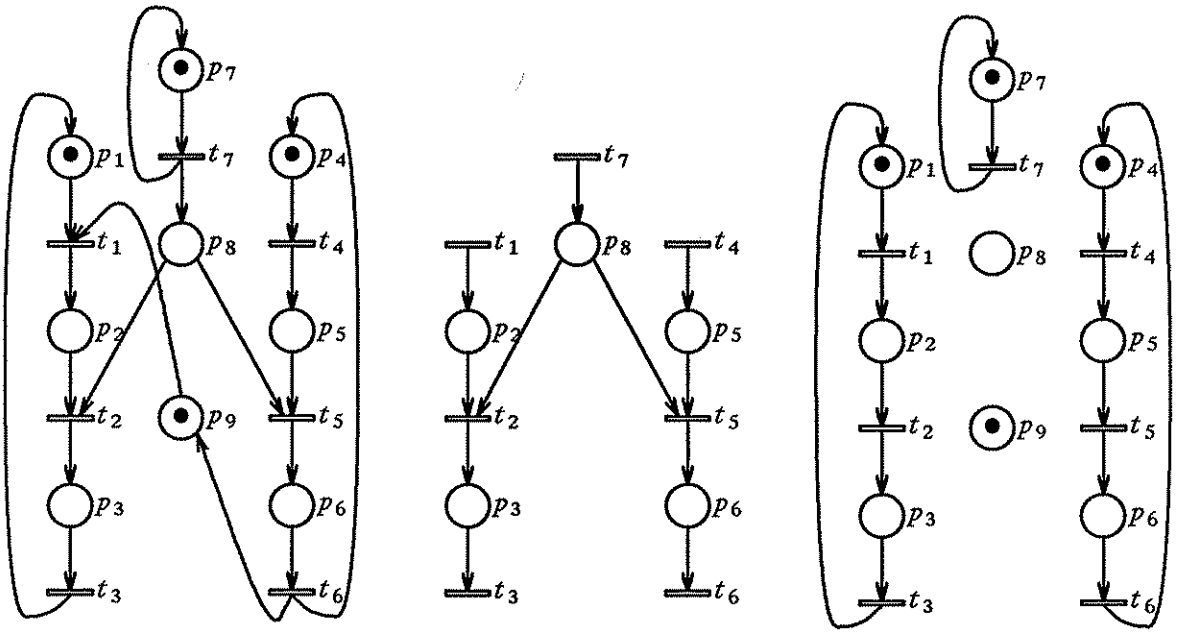
Figure 7-2: *Liveness Test For SM Program* C^1.

```
e = 0, a = 1: semaphore
cobegin                          — SM Program C²
        cycle                    — producer process
                V(e)             — produce an element and place it in the buffer
        endcycle
        //
        cycle                    — consumer process #1
                P(a)             — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V()              — allow the next consumer process to proceed
        endcycle
        //
        cycle                    — consumer process #2
                P()              — wait for this consumer's turn to come around
                P(e)             — remove an element from the buffer and consume it
                V(a)             — allow the next consumer process to proceed
        endcycle
coend
```

We construct N^2, M_0^2, U^2, T^2, and S^2 as shown in Figure 7-3. Since $T^2 \neq \lambda$, N^2 has no circuits unmarked in M_0^2. Since conflict place p_8 is the only node in its strong component, N^2 has no conflict circuits. Hence $live^2 = \textbf{true}$ and C^2 is deadlock-free.

Finally, we add the interconnection associated with the semaphore that allows the second consumer process to proceed.

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V(e)	t_1	P(a)	t_4	P()	p_8	e
		t_2	P(e)	t_5	P(e)	p_9	a
		t_3	V()	t_6	V(a)		



(a) *SM Net* N^2 (b) *Unmarked Subnet* U^2 (c) *Strong Components* S^2

$$T^2 = t_1 p_2 t_4 p_5 t_7 p_8 t_2 t_5 p_3 t_3 p_6 t_6$$

(d) *Topological Ordering* T^2

Figure 7-3: *Liveness Test For SM Program* C^2.

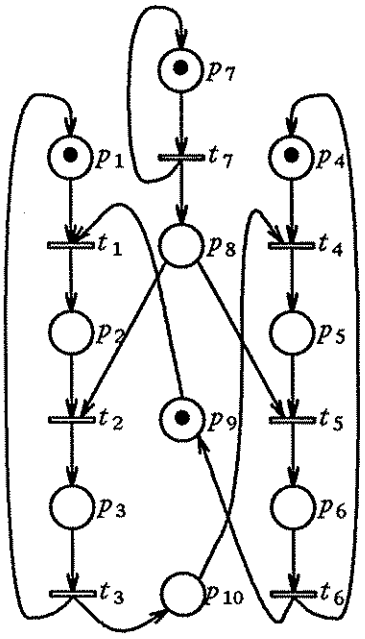
```
e = 0, a = 1, b=0: semaphore
cobegin                    — SM Program C³
        cycle              — producer process
                V(e)       — produce an element and place it in the buffer
        endcycle
        //
        cycle              — consumer process #1
                P(a)       — wait for this consumer's turn to come around
                P(e)       — remove an element from the buffer and consume it
                V(b)       — allow the next consumer process to proceed
        endcycle
        //
        cycle              — consumer process #2
                P(b)       — wait for this consumer's turn to come around
                P(e)       — remove an element from the buffer and consume it
                V(a)       — allow the next consumer process to proceed
        endcycle
coend
```

We construct N^3, M_0^3, U^3, T^3, and S^3 as shown in Figure 7-4. Since $T^3 \neq \lambda$, N^3 has no circuits unmarked in M_0^3. Since conflict place p_8 is the only node in its strong component, N^3 has no conflict circuits. Hence $live^3 = \textbf{true}$ and C^3 is deadlock-free.

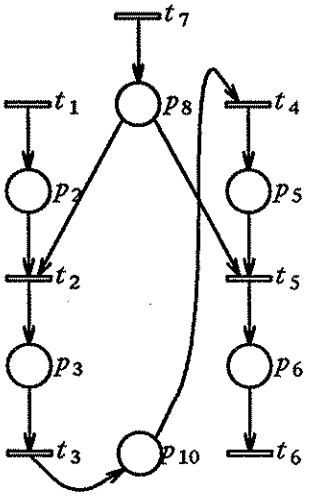
To this point we have reconstructed the unmarked subgraph, the topological ordering, and the strong components at each step and then applied the liveness test with no concern for efficiency. However, there are some cases where we can improve the algorithm.

Suppose that at step i we add or remove an interconnection associated with a semaphore whose initial value is greater than zero. Since this semaphore corresponds to a marked place, there is no need to recompute a new unmarked subnet or a new topological ordering. In this case, checking for unmarked circuits in the new *SM* net reduces to simply checking if the topological ordering of the old unmarked subnet is equal to λ. We saw an example of this in step 2 of the previous exam-

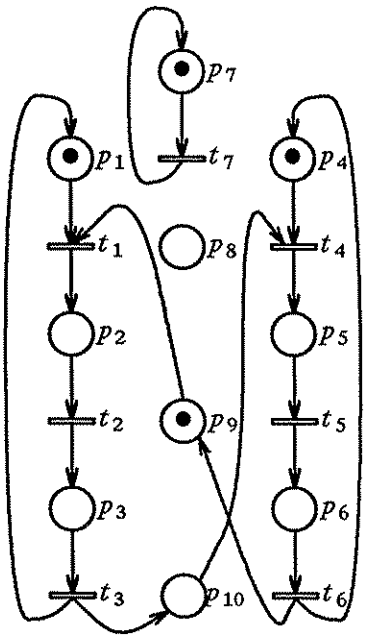
Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V(e)	t_1	P(a)	t_4	P(b)	p_8	e
		t_2	P(e)	t_5	P(e)	p_9	a
		t_3	V(b)	t_6	V(a)	p_{10}	b



(a) *SM Net* N^3



(b) *Unmarked Subnet* U^3



(c) *Strong Components* S^3

$$T^3 = t_1 p_2 t_7 p_8 t_2 p_3 t_3 p_{10} t_4 p_5 t_5 p_6 t_6$$

(d) *Topological Ordering* T^3

Figure 7-4: *Liveness Test For SM Program* C^3.

ple.

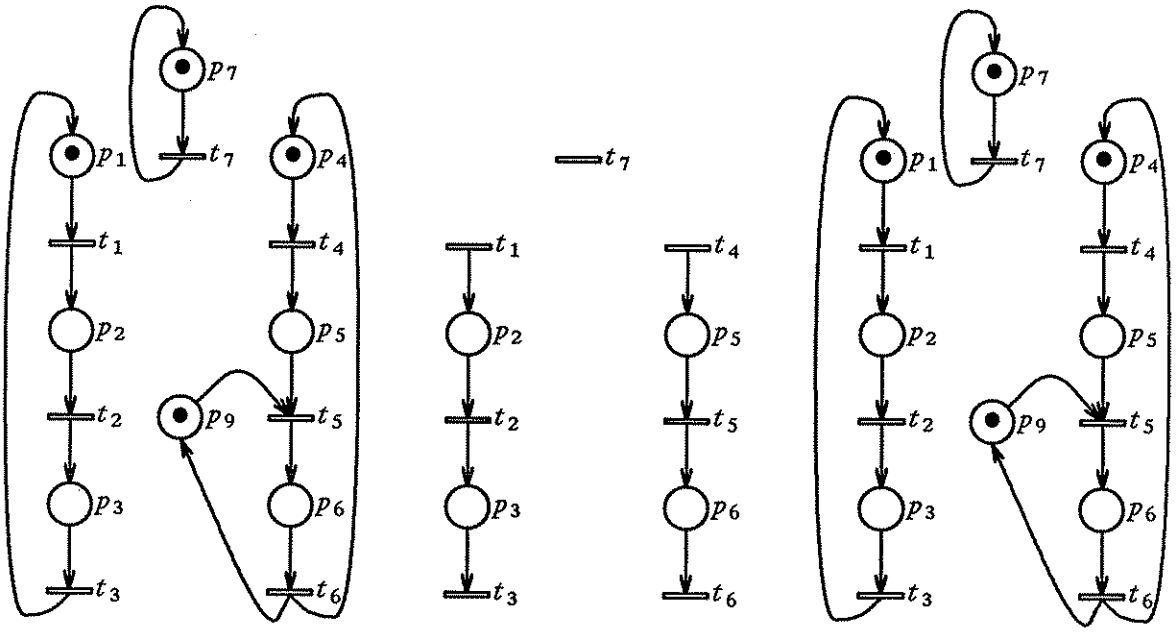
Suppose that at step i we add an interconnection associated with a semaphore σ. If all of the input and output transitions of its associated semaphore place p are in the same strong component S_k^i of the old SM net, then building the new strong components is easy. We simply add p and its input and output arcs to S_k^i to get S_k^{i+1}, and for all $j \neq k$, we set $S_j^{i+1} = S_j^i$. Further, if p is a marked conflict-free place, then $live^i$ implies $live^{i+1}$ and if p is a conflict place, then $live^{i+1} =$ **false**.

For example, suppose that we add the following interconnection to the initial SM program C^0.

```
a = 1: semaphore
cobegin                    — SM Program C1
        cycle              — producer process
                V()        — produce an element and place it in the buffer
        endcycle
        //
        cycle              — consumer process #1
                P()        — wait for this consumer's turn to come around
                P()        — remove an element from the buffer and consume it
                V()        — allow the next consumer process to proceed
        endcycle
        //
        cycle              — consumer process #2
                P()        — wait for this consumer's turn to come around
                P(a)       — remove an element from the buffer and consume it
                V(a)       — allow the next consumer process to proceed
        endcycle
coend
```

N^1, M_0^1, U^1, T^1, and S^1 are shown in Figure 7-5. Since semaphore place p_9 is marked, we simply use the unmarked subnet and topological ordering that we computed for C^0. Furthermore, since $t_6 = {}^\bullet p_9$ and $t_5 = p_9{}^\bullet$ are both in strong component $S_i^0 \in S^0$, we are able to construct S^1 by simply adding p_9 and its input

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V()	t_1 t_2 t_3	P() P() V()	t_4 t_5 t_6	P() P(a) V(a)	p_9	a



(a) *SM Net* N^1 (b) *Unmarked Subnet* U^1 (c) *Strong Components* S^1

$$T^1 = t_1 p_2 t_2 p_3 t_3 t_4 p_5 t_5 p_6 t_7$$

(d) *Topological Ordering* T^1

Figure 7-5: *Liveness Test for SM Net* C^1.

and output arc to S^0. Finially, since p is a marked conflict-free place with all of its input and output transitions in the same strong component of S^0 and since $live^0$ = **true**, we conclude that $live^1$ = **true** and that C^1 is deadlock-free.

As a final example, suppose that we then add an additional interconnection.

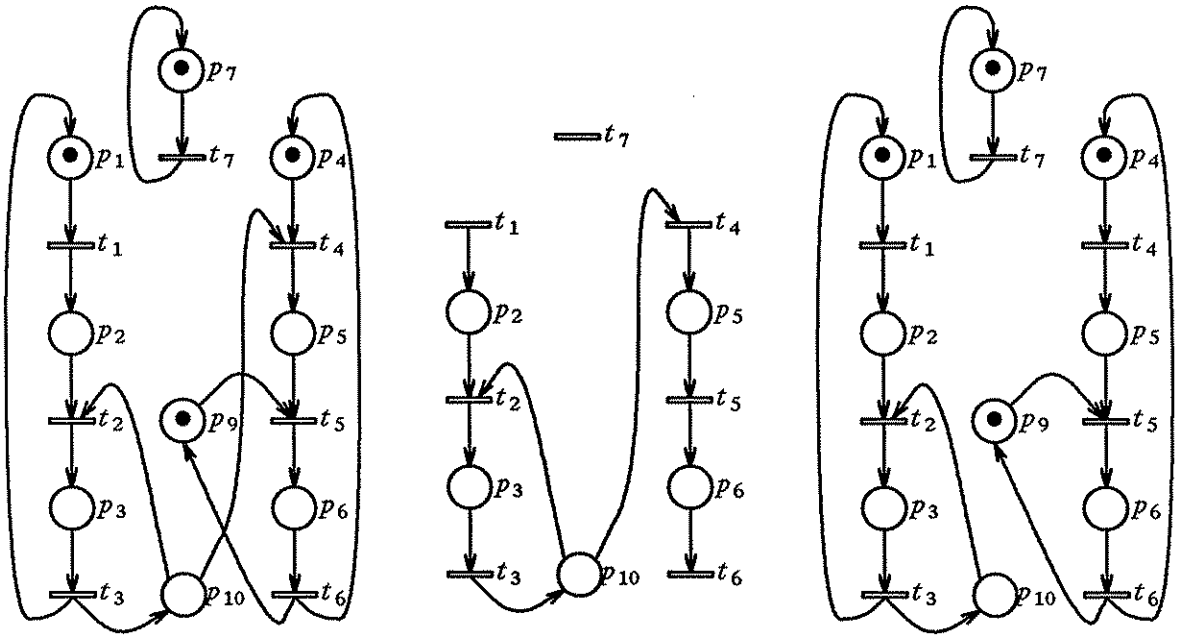
```
a = 1, b=0: semaphore
cobegin                    — SM Program C²
        cycle              — producer process
                V()        — produce an element and place it in the buffer
        endcycle
        //
        cycle              — consumer process #1
                P()        — wait for this consumer's turn to come around
                P(b)       — remove an element from the buffer and consume it
                V(b)       — allow the next consumer process to proceed
        endcycle
        //
        cycle              — consumer process #2
                P(b)       — wait for this consumer's turn to come around
                P(a)       — remove an element from the buffer and consume it
                V(a)       — allow the next consumer process to proceed
        endcycle
coend
```

N^2, M_0^2, U^2, T^2, and S^2 are shown in Figure 7-6. Because of the unmarked circuit $t_2 p_3 t_3 p_{10} t_2$, $T^2 = \lambda$, which implies that M_0^2 is not a live marking. Furthermore, conflict place p_{10} is not the only node in its connected component. This also implies that M_0^2 is not a live marking. Hence $live^2$ = **false** and C^2 is not deadlock-free.

7.3. Chapter Summary

We have demonstrated how static incremental deadlock detection might be performed in polynomial time on a subclass of *SM* semaphore programs. However, there are many interesting problems that remain to be investigated.

Legend							
Process 1		Process 2		Process 3		Semaphores	
trans	stmt	trans	stmt	trans	stmt	place	semaphore
t_7	V()	t_1	P()	t_4	P(b)	p_9	a
		t_2	P(b)	t_5	P(a)	p_{10}	b
		t_3	V(b)	t_6	V(a)		



(a) *SM Net* N^2 (b) *Unmarked Subnet* U^2 (c) *Strong Components* S^2

$$T^2 = \lambda$$

(d) *Topological Ordering* T^2

Figure 7-6: *Liveness Test For SM Program* C^2.

First, the analysis we outlined above does not make use of all of the available information; it essentially ignores statements with empty semaphore lists, and only considers those statements that have been associated with a semaphore that defines some interconnection. However, statements with empty semaphore lists contain information about the program that might be useful.

As a simple example, suppose that at the *ith* step in the program construction, all of the statements with empty semaphore lists are semaphore requests of the form $P()$. Clearly, unless interconnections are removed at a later step, the fully constructed program will not be deadlock-free. Thus, we need to investigate more sophisticated analysis techniques that make use all of the information about the program.

Second, we need to investigate techniques for performing static incremental deadlock detection on less restricted classes of concurrent systems than the *SM* programs considered here.

Third, suppose that at the *ith* step of the program construction the addition of an interconnection introduces the potential for deadlock. Is the interconnection defined at the *ith* step erroneous? Or is some collection of interconnections defined at earlier steps erroneous? Identifying these erroneous interconnections is an interesting problem that warrants further study.

CHAPTER 8

Conclusions

Our investigation into models of concurrent programs was motivated by a desire to find accurate and efficient techniques for statically and incrementally detecting deadlock potential in concurrent programs, before the programs are ever run. Towards this end, we studied issues related to accurate modeling and efficient analysis of models of concurrent programs.

We reported new results related to the accuracy of algebraic and geometric models and we reported new results on the efficient analysis of Petri net models. In the end, we used some of these results to develop an efficient technique for performing static incremental deadlock detection on a subclass of semaphore programs.

8.1. Summary of Results and Future Research

In Chapter 3, we described a new technique using predicate transformers for constructing a generalized deadlock predicate, the satisfiability of which is a necessary and sufficient condition for both partial and total deadlock. While the theory developed for the generalized deadlock predicate applies to all semaphore programs, we saw that the generalized deadlock predicate could only be constructed in practice for programs with a finite number of reachable states. We also pointed out that this restriction will no longer hold when and if a general solution to the finite models problem is found.

It seems likely that the modeling power of the generalized deadlock predicate can be increased to include programs that synchronize with conditional critical regions. Future research will explore this possibility.

In Chapter 4, we used theory from algebraic models and Petri net models to solve the finite models problem for a subclass of consumable resource semaphore programs called *SS* programs. While we were not able to solve the finite models for general semaphore programs, we did manage to increase the number of semaphore programs for which a solution to the finite models problem is known. Future research will center on a general solution to the finite models problem. We think that Petri net theory might provide some insight here. Failing to find a general solution, we will investigate other subclasses for which the finite models problem can be solved.

In Chapter 5, we derived a reachability theorem for Petri nets. We used this theorem to show that for a certain subclass of Petri nets, called *bounded inverse* nets, reachability can be decided by a simple enumeration of states, even though the number of reachable states is potentially infinite. Then we showed how this result can be used to analyze semaphore programs for reachability.

In Chapter 6, we derived necessary and sufficient conditions for the liveness of markings of a new subclass of Petri nets called *SM* Petri nets. We were then able to use these results to add *SM* nets to the list of Petri net subclasses for which the *deadlock–trap* property of [HACK72] is a necessary and sufficient condition for liveness. Then we showed how our results could be used to analyze a subclass of semaphore programs for deadlock potential.

Finally, in Chapter 7, we used the results of Chapter 6 to develop an efficient technique for performing static incremental deadlock detection on the subclass of general resource semaphore programs called *SM* programs.

Future research will center on developing efficient and accurate techniques for performing static incremental deadlock detection on less restrictive classes of concurrent programs. We are looking for synchronization constructs that are more general and that allow us to build models that capture more information about a program's dynamic behavior than say, semaphore operations. We are looking for analysis techniques that make full use of the information about a program's dynamic behavior that is available during the construction of the program. Finally, we are looking for ways to help the programmer find erroneous interconnections once deadlock potential has been detected in a partially constructed program.

8.2. Concluding Remarks

In the larger scheme of things, models such as algebraic, geometric, and Petri net models that support fully automatic and accurate analysis have rather limited modeling power. They are able to accurately capture the dynamic behavior of programs that consist simply of straightline synchronization primitives such as semaphore operations or conditional critical regions. We do not yet know how, or even if, we can automatically build and analyze accurate models of more general classes of concurrent programs.

Yet even with their limited modeling power, models such as algebraic, geometric, and Petri net models offer a surprising wealth of interesting problems, as we hope our work in this thesis has illustrated.

References

[AHO83] A. V. AHO, J. E. HOPCROFT and J. D. ULLMAN, in *Data Structures and Algorithms*, Addison-Wesley, 1983.

[ARAK77] T. ARAKI and T. KASAMI, Decidable Problems on the Strong Connectivity of Petri Net Reachability Sets, *Theoretical Computer Science 4*, (1977), 99-119.

[BLAZ84] J. BLAZEWICZ, D. P. BOVET and G. GAMBONI, Deadlock-Resistant Flow Control Procedures For Store-and-Forward Networks, *IEEE Transactions on Communications COM-32*, 8 (August 1984), 884-887.

[CARS84] S. D. CARSON, Geometric Models of Concurrent Programs, PhD Dissertation, University of Virginia, 1984.

[CHOW84] C. H. CHOW, M. G. GOUDA and S. S. LAM, An Exercise in Constructing Multi-Phase Communication Protocols, in *ACM Sigcomm Symposium*, 1984, 42-49.

[CLAR80] E. M. CLARKE, Synthesis of Resource Invariants for Concurrent Programs, *ACM Transactions on Programming Languages and Systems 2*, 3 (July 1980), 338-358.

[COFF71] E. G. COFFMAN, M. J. ELPHICK and A. SHOSHANI, System Deadlocks, *ACM Computing Surveys 3*, 2 (June 1971), 67-78.

[COMM71] F. COMMONER and A. W. HOLT, Marked Directed Graphs, *Journal of Computer and System Sciences 5*, (1971), 511-523.

[COMM72] F. COMMONER, Deadlocks in Petri Nets, Report CA-7206-2311, Massachusetts Computer Associates, Wakefield, Massachusetts, July 1972.

[CRES75] S. CRESPI-REGHIZZI and D. MANDRIOLI, A Decidability Theorem For a Class of Vector Addition Systems, *Information Processing Letters 3*, 3 (January 1975), 78-80.

[DATT84] A. DATTA and S. GHOSH, Synthesis of a Class of Deadlock-Free Petri Nets, *Journal of the ACM 31*, 3 (July 1984), 486-506.

[DIJK68] E. W. DIJKSTRA, Cooperating Sequential Processes, in *Programming Languages, (F. Genuys, ed.)*, Academic Press, 1968, 43-112.

[DIJK76] E. W. DIJKSTRA, in *A Discipline of Programming*, Prentice-Hall, 1976.

[FLON81] L. FLON and N. SUZUKI, Total Correctness of Parallel Programs, *SIAM Journal on Computing 10*, 2 (May 1981), 227-246.

[GENR73] H. GENRICH and K. LAUTENBACH, Synchronisationsgraphen, *Acta Informatica 2*, 2 (1973), 143-161.

[GOUD83] M. G. GOUDA, An Example For Constructing Communicating Machines by Step-Wise Refinement, in *Protocol Specification, Testing, and Verification III*, H. RUDIN and C. H. WEST (ed.), Elsevier Science Publishers B. V. (North Holland), 1983, 63-74.

[GRAB80] J. GRABOWSKI, The Decidability of Persistence for Vector Addition Systems, *Information Processing Letters 11*, 1 (1980), 20-23.

[GRIE79] W. GRIESE, Lebendigkeit in NSK-Petrinetzen, TUM-INFO-7906, Tech. Univ. Munchen, Februray 1979.

[GUNT81] K. D. GUNTHER, Prevention of Deadlocks in Packet Switched Data Transport Systems, *IEEE Transactions on Communications COM-29*, 4 (April 1981), 512-524.

[HABE72] A. N. HABERMANN, Synchronization of Communicating Processes, *Communications of the ACM 15*, 3 (March 1972), 171-176.

[HACK72] M. HACK, Analysis of Production Schemata by Petri Nets, Master's Thesis, Department of Electrical Engineering, Massachusetts Institute of Technology, 1972.

[HACK76] M. HACK, Decidability Questions for Petri Nets, Technical Report 161, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Massachusetts, June 1976.

[HAIL83] B. T. HAILPERN and S. S. OWICKI, Modular Verification of Computer Communication Protocols, *IEEE Transactions on Communications COM-31*, 1 (January 1983), 56-68.

[HANS81] P. B. HANSEN, *Basic Concepts of Network Programs*, Computer Science Department, University of Southern California, Los Angeles, 1981.

[HAVE68] J. W. HAVENDER, Avoiding Deadlock in Multitasking Systems, *IBM Systems Journal 7*, 2 (1968), 74-84.

[HERZ77] O. HERZOG, Automatic Deadlock Analysis of Parallel Programs, in *International Computing Symposium 1977*, E. MORLET and D. RIBBENS (ed.), 1977, 209-216.

[HERZ79] O. HERZOG, Static Analysis of Concurrent Processes for Dynamic Properties Using Petri Nets, in *Semantics of Concurrent Computation*, vol. 70, Springer-Verlag, 1979, 66-90.

[HOLT70] A. W. HOLT and F. COMMONER, Events and Conditions, in *Record of the Project MAC Conference on Concurrent Systems and Parallel Computation*,

Woods Hole, Massachusetts, June 1970, 3-52.

[HOLT72] R. C. HOLT, Some Deadlock Properties of Computer Systems, *Computing Surveys 4*, 3 (September 1972), .

[HOLT74] A. HOLT, Final Report for the Project 'Development of the Theoretical Foundations for Description and Analysis of Discrete Information Systems, CADD-7405-2011, Applied Data Research, Wakefield, Massachusetts, 1974.

[JANT79] M. JANTZEN and R. VALK, Formal Properties of Place/Transition Nets, in *Net Theory and Applications, Lecture Notes in Computer Science*, vol. 16, G. HOOS and J. HARTMANIS (ed.), Springer-Verlag, 1979, 165-212.

[KARP69] R. M. KARP and R. E. MILLER, Parallel Program Schemata, *Journal of Computer and System Sciences 3*, 2 (May 1969), 147-195.

[KELL76] R. M. KELLER, Formal Verification of Parallel Programs, *Communications of the ACM 19*, 7 (July 1976), 371-384.

[KELL77] R. M. KELLER, Generalized Petri Nets as Models for System Verification, Technical Report, University of Utah, 1977.

[KOSA82] S. R. KOSARAJU, Decidability of Reachability in Vector Addition Systems, in *ACM Symposium on Theory of Computing*, 1982, 267-281.

[LAMS79] A. LAMSWEERDE and M. SINTZOFF, Formal Derivation of Strongly Correct Concurrent Programs, *Acta Informatica 12*, (1979), 1-31.

[LAND78] L. LANDWEBER and E. ROBERTSON, Properties of Conflict-Free and Persistent Petri Nets, *Journal of the ACM 25*, 3 (July 1978), 352-364.

[LAUT74] K. LAUTENBACK and H. A. SCHMID, Use of Petri Nets for Proving Correctness of Concurrent Process Systems, in *Information Processing 74, Proceedings of the 1974 IFIF Congress*, North Holland, Amsterdam, August 1974, 187-191.

[LIEN76] Y. E. LIEN, Termination Properties of Generalized Petri Nets, *SIAM Journal on Computing 5*, 2 (June 1976), 251-265.

[LIPT76] R. LIPTON, The Reachability Problem Requires Exponential Space, Research Report 62, Yale University, New Haven, Connecticut, January 1976.

[MANC83] P. MANCARELLA and F. TURINI, A High Level Analysis Tool for Concurrent Programs, in *Proceedings of the 1983 Conference on Parallel Processing*, Columbus, Ohio, August 1983.

[MAYR81] E. W. MAYR, An Algorithm for the General Petri Net Reachability Problem, in *ACM Symposium on Theory of Computing*, 1981, 238-246.

[MAYR84] E. W. MAYR, An Algorithm for the General Petri Net Reachability Problem, *SIAM Journal on Computing 13*, 3 (August 1984), 441-460.

[MEMM78] G. MEMMI, Fuites dans les Reseaux de Petri, *Informatic Theorique 12*, (1978), 125-144.

[MERL80] P. M. MERLIN and P. J. SCHWEITZER, Deadlock Avoidance in Store and Forward Networks - I: Store and Forward Deadlock, *IEEE Transactions on Communications COM-28*, 3 (March 1980), 345-354.

[MURA77] T. MURATA, Circuit Theoretic Analysis and Synthesis of Marked Graphs, *IEEE Transactions on Circuits and Systems CAS-24*, 7 (July 1977), 400-405.

[O'HA86] D. R. O'HALLARON and P. F. REYNOLDS JR., A Generalized Deadlock Predicate, *to appear in Information Processing Letters*, , 1986.

[OWIC76] S. OWICKI and D. GRIES, Verifying Properties of Parallel Programs: An Axiomatic Approach, *Communications of the ACM 19*, 5 (May 1976), 279-284.

[OWIC82] S. OWICKI and L. LAMPORT, Proving Liveness Properties of Concurrent Programs, *ACM Transactions on Programming Languages and Systems 4*, 3 (July 1982), 455-495.

[PETE81] J. L. PETERSON, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, 1981.

[PFAL77] J. L. PFALTZ, *Computer Data Structures*, McGraw-Hill, 1977.

[REYN79] P. F. REYNOLDS, JR., Parallel Processing Structures: Languages, Schedules, and Performance Results, Ph.D. Thesis, University of Texas, Austin, 1979.

[SIDH82] D. P. SIDHU, Protocol Design Rules, in *Protocol Specification, Testing, and Verification* , C. SUNSHINE (ed.), North Holland, 1982, 63-74.

[SNYD84] L. SNYDER, Parallel Programming and the Poker Programming Environment, *IEEE Computer 17*, 7 (July 1984), 27-37.

[TAYL83] R. N. TAYLOR, A General Purpose Algorithm for Analyzing Concurrent Programs, *Communications of the ACM 26*, 5 (May 1983), 362-376.

[THIA84] P. THIAGARAJAN and K. VOSS, A Fresh Look at Free Choice Nets, *Information and Control 62*, (May 1984), 85-113.

[WIMM84] W. WIMMER, Using Barrier Graphs for Deadlock Prevention in Communication Networks, *IEEE Transactions on Communications COM-32*, 8 (August 1984), .

[YAMA81] H. YAMASAKI, On Weak Persistency of Petri Nets, *Information Processing Letters 13*, 3 (December 1981), 94-97.

[YAMA84] H. YAMASAKI, Normal Petri Nets, *Theoretical Computer Science 31*, (1984), 307-315.

[YU82] Y. YU and M. G. GOUDA, Deadlock Detection for a Class of Communicating Finite State Machine, *IEEE Transactions on Communications COM-30*, 12 (December 1982), 2514-2518.

[ZAFI80] P. ZAFIROPULO, C. H. WEST, H. RUDIN, D. D. COWAN and D. BRAND, Towards Analyzing and Synthesizing Protocols, *IEEE Transactions on Communications COM-28*, 4 (April 1980), 651-660.