DEADLOCK DETECTION AND AVOIDANCE FOR A CLASS OF MANUFACTURING SYSTEMS

A Thesis Presented to

The Faculty of the

Fritz J. and Dolores H. Russ
College of Engineering and Technology

Ohio University

In Partial Fulfillment

of the Requirement for the Degree

Master of Science

by

Tariq Nadeem Faiz

March, 1996

ACKNOWLEDGMENTS

I would like to thank my advisor, Dr. Robert P. Judd. His interest in manufacturing research, brilliant insights and painstaking attention to detail are instrumental in the completion of this Thesis. It has been a pleasure working with him and I am sincerely grateful for his help.

I would like to thank Dr. Constantine Vassiliadis, Dr. Joseph H. Nurre and Dr. Luis Rabelo for agreeing to be members of my thesis committee.

I would again thank my parents for their encouragement, support and patience. I thank my fiancee Bethany Klopp for her encouragement, Paul Deering for his help with Chapters 1 and 7, and my roommates Craig Moore, Robert VanWagenen, Steve Grimm, who would refresh me with stimulating conversations.

TABLE OF CONTENTS

Page

LIST OF TABLES

Page

LIST OF FIGURES

Page

CHAPTER 1

INTRODUCTION

Deadlock is a phenomenon which occurs in Flexible Manufacturing Systems when multiple processes compete for the use of limited resources. In this research a method for detection of deadlock and its avoidance has been developed.

Consider the example of a flexible manufacturing workstation consisting of a Robot and a CNC Mill machine, as shown in Figure 1.1.

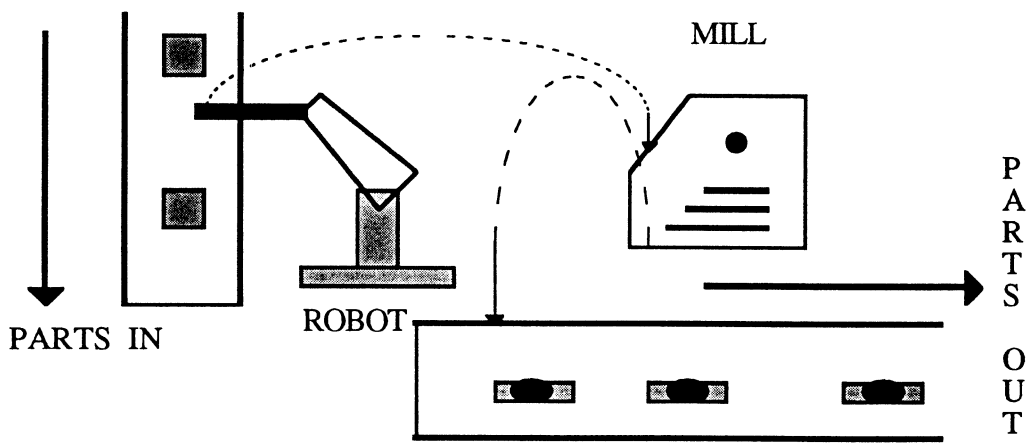

Figure 1.1. Example of a manufacturing system.

Parts arrive on a conveyor belt ("Parts In") and are picked up immediately by the robot for transportation to the Mill. When a part has been processed by the Mill it is transported by the robot to the exit conveyor ("Parts Out"). Assume now that a part arrives on the conveyor belt and is picked up and transferred to the Mill. Before this part can be processed completely in the Mill, another part arrives on the conveyor. The robot picks the part up and then waits for the Mill to be free. The Mill will eventually complete its machining operations and the robot will be called upon to transfer the part to the exit

conveyor. However the robot is busy—it has a part which needs to be transferred to the Mill. The Mill is busy—it needs the robot to transfer a part out. Deadlock has occurred.

1.1 GOALS

In this thesis we seek to develop a theory for the detection and avoidance of deadlock in a class of manufacturing systems. The thesis has the following goals:

1. To develop formalisms for the detection of deadlock in a class of flexible manufacturing systems.
2. To develop formalisms for the avoidance of deadlock so as to guarantee a deadlock-free system.
3. To develop a computer program which implements the deadlock detection and avoidance method developed.
4. To test the method developed on a number of example systems.

1.2 LITERATURE SEARCH

The problem of deadlock has been addressed by a number of researchers. In this section a survey of research in the field is presented. A comparison with the method developed in this thesis is presented for every research paper surveyed.

Wysk, Joshi, Yang [1] have used directed graphs to detect deadlock in Flexible Manufacturing Systems. In their approach, called a Deadlock Detection Procedure(DDP), a structure called a Wait Relation Graph(WRG) is used to model part flow and resource-operation relations. Briefly, a WRG is a directed graph with nodes representing resources and arcs operations. The WRG is developed at each state and examined for deadlock. Appropriate avoidance measures are then taken. This thesis develops on the method

presented in this paper, and WRGs are detailed in Chapter 2. The paper identifies two levels of deadlock. In the first theorem sufficient conditions for the occurrence of first level deadlock are explained. Briefly these are: the existence of at least one circuit; number of active jobs in this circuit must equal the number of arcs; the number of resources in the circuit must equal the number of arcs. The paper contains a detailed description of a matrix multiplication method used to detect all circuits. The second level deadlock is then described. This occurs from interaction between circuits. The paper describes a method to convert a WRG to a second-level graph and detect all second-level circuits.

Cho, Kumaran, Wysk [2] have presented a graph-theoretic deadlock detection and resolution procedure. A "system status graph" is used to represent part routings of all parts of the system. It is updated whenever a part movement occurs. The graph is then analyzed for deadlock. Two types of system deadlocks--part flow deadlock" and "impending part flow deadlock--are detected using the system status graph. A part flow deadlock is a situation in which further part movements are impossible. The existence of "a simple bounded circuit" is "a sufficient and necessary condition for part flow deadlock". Part flow deadlock is resolved by moving a part to a temporary storage buffer and then sequentially moving the other parts. A second scheme of deadlock avoidance can also be employed if buffers are full or not available. Here the system is checked for part flow deadlock before the part moves to its next destination. The deadlock causing transition is then inhibited. No guidelines on when each method would be suitable are presented. The concept of "nonsimple bounded circuit" is introduced: "a non simple bounded circuit is a necessary condition for impending part flow deadlock". Both cases of empty and non-empty common nodes are considered. A heuristic scheme for analyzing deadlock in nonsimple bounded circuits is also presented.

Zhou and DiCesare [3] have worked on the shared resource allocation problem. Two essential resource sharing concepts in the context of Petri Nets are developed. The

first is a structure called a Parallel Mutual Exclusion (PME) proposed for a resource shared by different independent processes. It is represented as a tuple that comprises an initially marked place and a set of transition pairs. The shared resource place models the availability of the shared resource, and each transition pair defines a process. A theorem on Petri Nets containing PMEs is presented. Briefly, the net is live, bounded and reversible if the shared resource is added as a PME. A Sequential Mutual Exclusion (SME) models a resource shared by sequentially related processes. It can be visualized as a sequential composition of several PMEs with the same shared resource place. The liveness and reversibility of a net containing a SME can be affected by an inappropriate distribution of initial tokens. A concept of token capacity is proposed. A second theorem provides sufficient conditions for a Petri Net that contains a SME to be live, bounded and reversible. Briefly, the number of initial tokens should be limited to the token capacity of the SME.

Banaszak and Krogh [4] have developed an algorithm to avoid deadlock in Flexible Manufacturing Systems. Their Deadlock Avoidance Algorithm (DAA) is a restriction policy for constraining real-time resource allocation options. Deadlock is identified as being caused by circular wait relations between resources. The DAA is a feedback policy that uses the current states of the resources and the known operation sequences for the active jobs to inhibit requests for resources only when they will potentially lead to deadlock conditions. DAA partitions the production sequence for a process into subsequences or zones. Unshared zones correspond to production steps using unshared resources. Shared zones correspond to production steps using shared resources. Two DAA rules are presented. Rule DAA1 allows a token to enter a new zone in the production sequence only when the capacity in the unshared subzone of the zone exceeds the number of jobs currently in the zone. Rule DAA2 assures that if a shared resource is being requested by the job, all of the shared resources in the remainder of the zone are available at that time.

Hsieh and Chang [5] have developed a method to synthesize a deadlock free controller based on the Petri Net(PN) formalism. They start by constructing independent PNs to represent the manufacturing processes of individual jobs and individual resources. These jobs and resource subnets are merged into a Petri Net representing the entire system and control places added. The deadlock avoidance problem is then formulated as finding a sequence of transition firings which will keep the Petri Net live and achieve a high resource utilization. The sequence of transitions is called a control action. A set of theorems which describe conditions for synthesis of deadlock-free control actions is presented. Concepts such as "minimum resource requirement" are introduced and used in determining when it is possible to synthesize a control policy which keeps the net live. In addition to the dispatching policy, a "Sufficient Validity Test" is introduced. This evaluates in a heuristic way when a control policy is valid and when it must be replaced. A replacement procedure to modify an invalid control action and then apply the sufficient validity test is described. A finite number of such iterations is needed to find a valid, replacement, control policy. The Deadlock Avoidance Controller(DAC)--consisting of the dispatching policy, sufficient validity test and replacement procedure--is applied at every state to determine the next valid state.

On comparing the deadlock detection and avoidance method developed in this thesis with current research, we found that the method is more reliable and achieves better resource utilizations compared to PME, SME, DAA and DAC methods. Another improvement was that all computations for deadlock detection and avoidance are done off-line. There is no diversion of computation resources for any deadlock related calculations while the process is running.

1.3 OUTLINE OF THESIS

The next chapters will present the deadlock detection and avoidance theory, the computer program developed, and iluustrative examples. Chapter 2 presents formalisms developed for the detection and avoidance of deadlock in manufacturing systems. Assumptions for the class of manufacturing systems considered, definitions and theorems are present in this chapter. Chapters 3 through 5 describe the development of a computer program to detect and avoid deadlock in a FMS. Chapter 3 describes algorithms used in the creation of the Wait Relation Graph of the system. Chapter 4 describes algorithms used in detection of primary circuits. The algorithms in Chapter 5 are used in detection of higher order circuits. Chapter 6 contains results obtained from using the computer program to analyze various example FMSs. Conclusions follow in Chapter 7.

CHAPTER 2
THE DEADLOCK FREE SOLUTION

In this chapter a solution to the problem of deadlock in manufacturing systems is presented. The chapter starts with an illustrative example, one which is representative of the type of manufacturing systems being modelled in this research. Assumptions regarding the nature of manufacturing systems being modelled are then presented. After a brief section on terminology, the Wait Graph Relation(WRG) technique of modelling a manufacturing system is introduced. Important WRG structures such as paths and circuits are defined. The occurence of deadlock is illustrated using Wait Relation Graphs. Next, the formalism proposing the deadlock free solution is presented. The chapter concludes with a generalized formalism, one which can create the deadlock free solution to any manufacturing system.

2.1. ILLUSTRATIVE EXAMPLE

In this section, a hypothetical example typical of the manufacturing systems modelled in this research is presented.

Dies Incorporated is a small company in Athens, Ohio. They manufacture four different dies for the truck industry; their main customer is Kenworth Truck Company in Chillicothe, Ohio. The process plan for manufacture of the four dies is as follows:

```
Die 1 - Robot, Mill,    Robot,    Lathe,    Robot.
Die 2 - Robot, Lathe,   Robot.
Die 3 - Robot, Drill,   Robot.
Die 4 - Robot, Mill,    Robot,    Drill,    Robot.
```

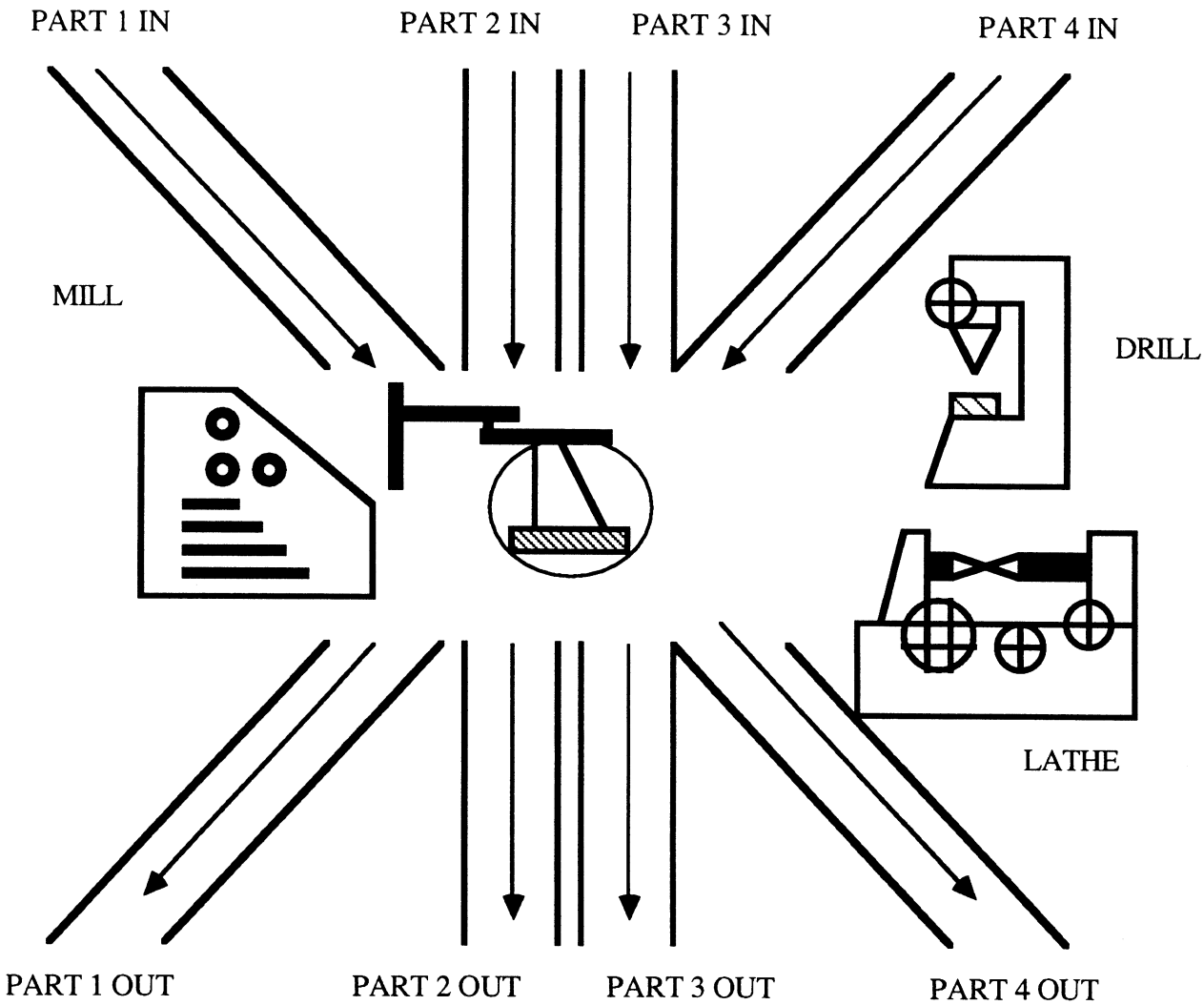

Figure 2.1.Dies Incorporated--4 processes sharing 4 resources.

The resources in this system are Robot, Drill, Mill and Lathe. There is one unit of each type of resource. The manufacture of each of the four dies can proceed concurrently, sharing the available resources. If a resource required by a specific operation is busy, the part will wait until the resource is free again. Thus, we can have multiple dies of the same type being manufactured, and have different dies being manufactured concurrently. This system is representative of the kinds of manufacturing systems modelled in this research.

2.2. ASSUMPTIONS

The following assumptions are made regarding the nature of the manufacturing system being modelled:

1. An operation uses just one resource.
2. There is one unit of every resource in the system.
3. There is no branching of operations in a process plan. Any operation is always preceeded, and/or suceeded by a single operation.
4. An operation can only process one part at any time.
5. All operations take a finite time for completion.
6. There are a finite number of operations in the process plan.

2.3 TERMINOLOGY

The following terminology will be used in this research:

R	Set of resources in the system.
Q	Set of products to be manufactured.
Oper(q)	For each product $q \in Q$, the operation sequence Oper(q) = $\{r_a, r_b, r_c, ..., r_m\}$ is an ordered list of resources,

which defines the order that the resources are required by process q.

2.4. THE WAIT RELATION GRAPH REPRESENTATION OF SYSTEMS

In this research, the Wait Relation Graph technique of modelling manufacturing systems is adopted. The present section contains a series of definitions introducing Wait Relation Graphs and related concepts. These are followed by the Wait Relation Graph representation of the Dies Incorporated system presented in Section 2.1.

2.4.1 DEFINITIONS

Wait Relation Graph

The Wait Relation Graph $G = (R, A)$ is a digraph of vertices and arcs. R is the set of vertices; A is the set of arcs. Each vertex represents a resource in the system. The arcs represent the operations in the system. An arc a is drawn between resources r_1 and r_2, if r_2 immediately follows r_1 in at least one operation sequence Oper(q), $q \in Q$.

Head function

Given an arc $a \in A$, head(a) = r, if arc a is directed from resource v to resource r.

Tail function

Given an arc $a \in A$, tail(a) = v, if arc a is directed from resource v to resource r.

Subgraph

A subgraph $G_1 \subset G$ is a graph where $R_1 \subset R$, $A_1 \subset A$ and tail(a) $\in R_1$, head(a) $\in R_1$, $\forall\, a \in A_1$.

Path

A *path* $P=(R, A)$ is a subgraph of G such that all elements of R and A can be ordered as

$$r_1a_1r_2a_2r_3a_3\cdots a_{n-1}r_n,$$

where $r_i = \text{tail}(a_i)$ and $r_{i+1} = \text{head}(a_i)$.

Simple Path

A Simple Path $P = (R, A)$ is a path where all the elements in the ordered list are distinct.

Closed Path

A Closed Path $P = (R, A)$ is a simple path where the ordered elements form a loop. Arcs and vertices may be traversed more than once in the loop.

Circuit

A circuit is a closed path.

Primary Circuit

A Primary Circuit $C=(R, A)$ is a closed path where each resource can be at the head or tail of only one arc.

Union of circuits

A union of circuits $C_1 = (R_1,A_1)$, $C_2 = (R_2,A_2)$ is a single closed path $C = (R, A)$ where $R = (R_1 \cup R_2)$ and $A = (A_1 \cup A_2)$. C is denoted as $C = C_1 \cup C_2$

Committed Arc

An arc a is *committed* if any one operation represented by arc a is processing a part.

Committed Resource

A resource r is *committed* if any arc a is committed where tail(a) = r.

Free Resource

A resource r is *free* if it is not committed.

Busy Resource

A resource r is *busy* if it is processing a part.

Idle Resource

A resource is *idle* if it is not processing a part.

A distinction exists between committed, free, busy and idle resources. A resource may be busy and not committed. This distinction arises only for those resources required at the end of a process. Consider the manufacturing system M whose process plan is as follows:

```
Part 1      Robot Mill  Robot
Part 2      Robot Lathe
```

The Wait Relation Graph is depicted in Figure 2.2.

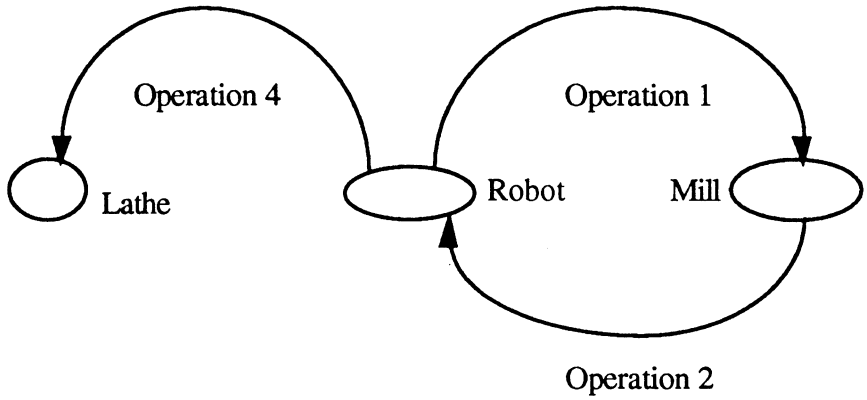

Figure 2.2. Wait Relation Graph for system *M*.

To illustrate the distinction between committed, free, busy and idle resources, assume:

1. The Robot is transporting part 1 to the Mill.
2. The Lathe is processing part 2.
3. The Mill is not processing any part.

In this state, the Robot is committed. However the Lathe being the last resource required in process 2 is not committed to any arc. We say it is free, although it is busy. The remaining resource--namely the Mill--is not processing any part and is an idle and free resource.

State of a manufacturing system

The state *s* of a manufacturing system is the current assignment of operations to resources.

The set S

S is the set of all admissable states *s* for the manufacturing system.

Commitment

The commitment of arc a in state s is defined as

$$\text{Comm}(a, s) = \begin{cases} 1 & \text{if } a \text{ is a committed arc} \\ 0 & \text{if } a \text{ is not a committed arc} \end{cases}$$

Commitment of a subgraph

The commitment of a subgraph $G_1 = (R_1, A_1)$ in state s is the sum of the commitment of every arc $a \in A_1$ and is expressed as

$$\text{Comm}(G_1, s) = \sum_{a \in A_1} \text{Comm}(a, s). \tag{2-1}$$

Capacity

Capacity of a subgraph $G_1 = (R_1, A_1)$ is the number of resources or vertices in G_1. Capacity of a subgraph is abbreviated $\text{Cap}(G_1)$. (Note that $\text{Cap}(G_1) = \text{Cardinality}(R_1)$.)

Slack

Slack of a circuit $C = (R, A)$ in state s is the difference between the capacity of the circuit and the commitment of all its arcs in s and is expressed as

$$\text{Slack}(C, s) = \text{Cap}(C) - \text{Comm}(C, s). \tag{2-2}$$

Propagation

Propagation is the transfer of a part from one operation to the next. In a propagation, the operation the part is currently in is completed, the resource occupied is freed, the part is transfered to the next operation and the next operation is started. The resource needed by the operation is occupied and busy.

This concludes the section on defintions. We now consider an example of a manufacturing system. All circuits in the system will be listed.

2.4.2 EXAMPLE

The Wait Relation Graph representation of the manufacturing system at the Dies Incorporated company is shown in Figure 2.3.

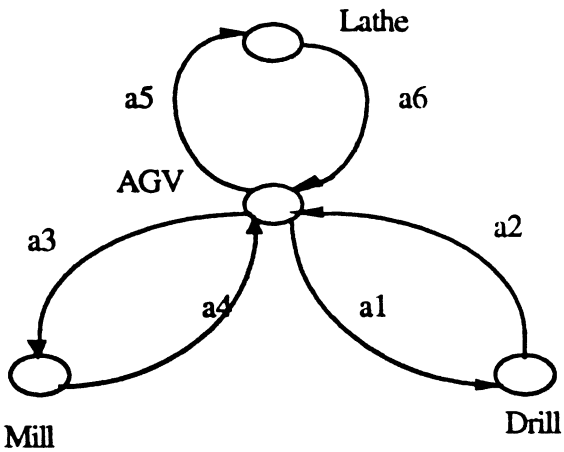

Figure 2.3. Wait Relation Graph of Dies Incorporated.

For the Wait Relation Graph, the following sets are defined:

1. R = { Robot, Drill, Mill, Lathe}.
2. $A = \{a_1, a_2, a_3, a_4, a_5, a_6\}$.

There are three primary circuits in the Wait Relation Graph. These are:

1. C1: ({Robot, Drill}, $\{a_2 , a_1\}$), where the elements are ordered,

   Robot a_1 Drill a_2 Robot

2. C2: ({Robot, Mill}, $\{a_3 , a_4\}$), where the elements are ordered,

   Robot a_3 Mill a_4 Robot; and

3. C3: ({Robot, Lathe}, $\{a_5 , a_6\}$), where the elements are ordered,

Robot a_5 Lathe a_6 Robot.

The following are unions of the primary circuits:

1. $C_4 : C_1 \cup C_2$, where the elements are ordered,

   Robot a_1 Drill a_2 Robot a_3 Mill a_4 Robot;

2. C_5: $C_2 \cup C_3$ where the elements are ordered,

   Robot a_3 Mill a_4 Robot a_5 Lathe a_6 Robot;

3. C_6: $C_1 \cup C_3$ where the elements are ordered,

   Robot a_1 Drill a_2 Robot a_5 Lathe a_6 Robot; and

4. C_7: $C_1 \cup C_2 \cup C_3$ where the elements are ordered,

   Robot a_1 Drill a_2 Robot a_3 Mill a_4 Robot a_5 Lathe a_6 Robot.

2.5. ANALYSIS OF CIRCUITS IN WAIT RELATION GRAPHS

This section presents the formalisms which develop a solution to the problem of deadlock. The theory first considers structurally simple Wait Relation Graphs. Theorems 1 and 2 consider systems whose WRGs consist of a single circuit. In theorems 3 and 4, WRGs considered consist of two circuits intersecting in a single vertex and along a simple path, respectively. Theorems 5 and 6 consider larger systems. Theorem 5 examines the case where an existing WRG is altered by the addition of a single circuit intersecting the existing Graph in a vertex. In Theorem 6, the WRG is altered by the addition of a single circuit intersecting the existing Graph along a simple path. The section concludes with a final theorem for general systems. Here the deadlock-free solution to a general system is created using the results presented in the previous six theorems.

The most basic type of deadlock occurs when all the resources on a simple circuit are committed. Figure 2.4 depicts an example of a primary circuit. The graph is representative of the systems considered in Theorems 1 and 2.

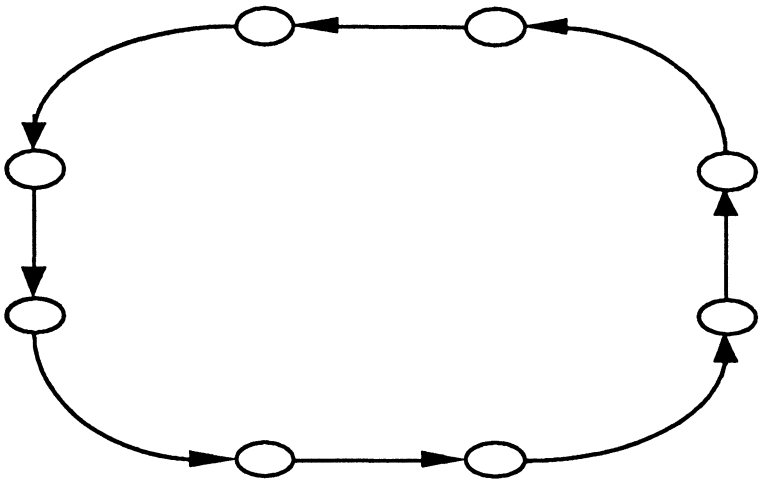

Figure 2.4. System whose WRG is a primary circuit.

Theorem 1 describes the occurence of deadlock in a manufacturing system whose Wait Relation Graph consists of a single primary circuit.

Theorem 1. Let the Wait Relation Graph G of a manufacturing system consist of a single primary circuit $C = (R, A)$. C is deadlocked in state s, if

$$\text{slack}(C, s) = 0. \tag{2-3}$$

Proof. In state s, (2-3) implies $\text{Comm}(C, s) = \text{Cap}(C)$. Hence, each arc in C is committed. This means each resource must wait until the next one is free before it can propagate its part. Hence, propagation is not possible, and state s is deadlocked. ■

One way to prove that a state s is not deadlocked is to determine a series of propagations which will remove all parts from the system. The following lemmas formalize this concept, laying the foundation for the proofs.

Lemma 1. Let C be a closed path in a Wait Relation Graph G of a system. Let s_0 be the initial state in C. For any state s_i reached after a finite number of propagations

$$\text{slack}(C, s_i) \geq \text{slack}(C, s_0).$$

Proof. As propagation occurs over C, no new parts enter C. The number of committed arcs in C either decreases--if parts are completed and removed from the system--or is unchanged. In either case, slack of circuit C cannot increase. ■

Lemma 2. Let s_i be any state in a system. Let no new parts enter the system. If deadlock does not exist, parts in the system can be removed in a finite number of propagations.

Proof. There are a finite number of operations in any process. Assuming there is no deadlock, after a finite number of propagations a part will complete all operations in its process plan and exit the system. As no new parts enter the system, the number of parts in the system will decrease as a part exits the system. After a finite number of operations, all parts will complete their operations and exit the system. ■

The following theorem proposes a deadlock free solution to a system whose Wait Relation Graph consists of a primary circuit.

Theorem 2. Let the Wait Relation Graph G of a manufacturing system consist of a primary circuit C. C is deadlock-free if

$$\text{slack}(C, s) > 0, \forall\, s \in S. \tag{2-4}$$

Proof. Let $\text{slack}(C, s_0) > 0$ for some state $s_0 \in S$. In state s_0, at least one arc is not committed, implying there is one free resource r in the system. Propagation can occur, specifically, on the operation represented by the arc a, where $\text{tail}(a)=r$. Let s_1 be the state of the system after the propagation. By Lemma 1

$$\text{slack}(C, s_1) \geq \text{slack}(C, s_0) > 0.$$

This process can be repeated until all parts exit the system. ■

Theorem 2 is applicable to a system whose Wait Relation Graph consists of a single circuit. Wait Relation Graphs in general consist of circuits intersecting in one or more vertices. Henceforth, we will consider such systems. We start by considering the most basic of such systems--those comprised of the union of two primary circuits. Figure 2.5 depicts the graph of such a system. Here two circuits, C_1 and C_2, intersect in a single vertex v. There is no assumption on the number of vertices and arcs in the circuits.

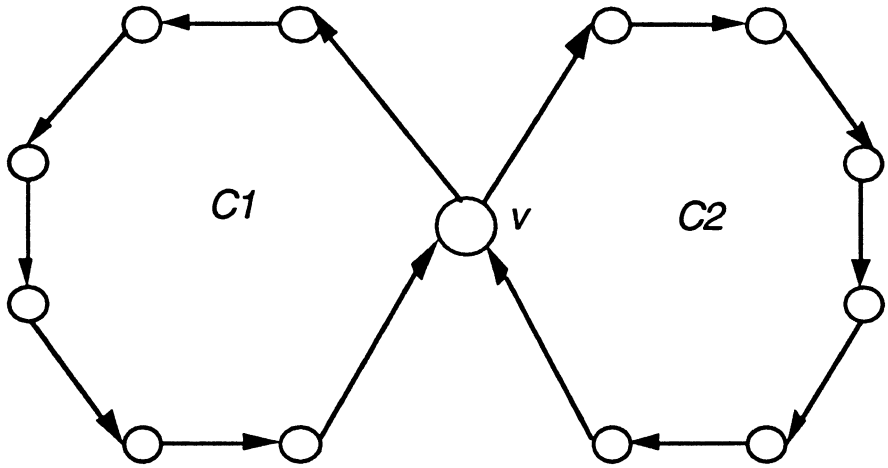

Figure 2.5. System of two circuits interacting in a single vertex.

In the next lemma the slack of the union of two circuits interesecting in a single vertex will be expressed in terms of the slacks of the individual circuits. Figure 2.5 will serve as a reference for this lemma.

Lemma 3. The slack of a circuit $C = C_1 \cup C_2$, where C_1 ,C_2 intersect in a single vertex v, is given by

$$\text{slack}(C_1 \cup C_2, s) = \text{slack}(C_1, s) + \text{slack}(C_2, s) - 1.$$

Proof. As C_1 ,C_2 intersect in a single vertex,

$$\text{Cap}(C) = \text{Cap}(C_1) + \text{Cap}(C_2) - 1. \tag{2-5}$$

As there are no common arcs between C_1 ,C_2,

$$\text{Comm}(C, s) = \text{Comm}(C_1, s) + \text{Comm}(C_2, s). \tag{2-6}$$

Combining (2-2), (2-5), and (2-6), we get

$$\text{slack}(C, s) = \text{Cap}(C_1) + \text{Cap}(C_2) - 1 - \text{Comm}(C_1, s) - \text{Comm}(C_2, s),$$

which simplifies to give

$$\text{slack}(C_1 \cup C_2, s) = \text{slack}(C_1, s) + \text{slack}(C_2, s) - 1. \quad \blacksquare$$

Two circuits could also intersect along a path. Figure 2.6 depicts the WRG of a system of two circuits interacting along a simple path P. Lemma 4 analyzes the system and presents a result on the slack.

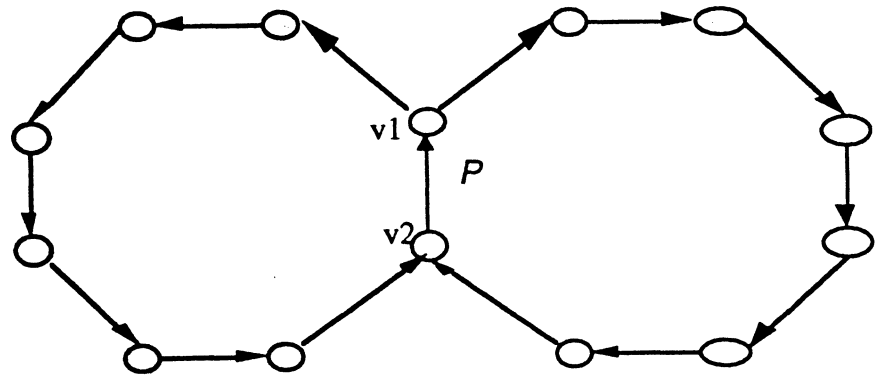

Figure 2.6. Two circuits intersecting along a simple path P.

Lemma 4. The slack of a circuit $C = C_1 \cup C_2$, where C_1 and C_2 intersect along a simple path P which begins at v_1 and ends at v_2 is given by

$$\text{slack}(C_1 \cup C_2, s) = \text{slack}(C_1, s) + \text{slack}(C_2, s) - \text{Cap}(P) + \text{Comm}(P, s).$$

Proof. As C_1 and C_2 intersect in a simple path, capacity of the union of C_1 and C_2 is given by

$$\text{Cap}(C) = \text{Cap}(C_1) + \text{Cap}(C_2) - \text{Cap}(P). \tag{2-7}$$

As there are common arcs between C_1 and C_2, commitment of the union of C_1 and C_2 is given by

$$\text{Comm}(C, s) = \text{Comm}(C_1, s) + \text{Comm}(C_2, s) - \text{Comm}(P,s). \tag{2-8}$$

Combining (2-2), (2-7), and (2-8), we obtain

$$\text{slack}(C_1 \cup C_2, s) = \text{slack}(C_1, s) + \text{slack}(C_2, s) - \text{Cap}(P) + \text{Comm}(P, s). \quad \blacksquare$$

Lemma 5 considers a system comprising two circuits C_1 and C_2 which intersect each other. A result on the slack of circuit C_2, due to a propagation on circuit C_1, is presented.

Lemma 5. Let C_1 and C_2 be two circuits which intersect each other. In state s_0, let a part propagate in C_1. Let s_1 be the state reached after propagation. Then,

$$\text{slack}(C_2, s_1) \geq \text{slack}(C_2, s_0)\text{-}1$$

Proof. In state s_0, a part could propagate from $C_1 = (R_1, A_1)$ to $C_2 = (R_2, A_2)$, increasing $\text{Comm}(A_2, s_1)$ by 1. Hence, $\text{slack}(C_2, s_1)$ decreases by at most 1. If no propagation occurs into C_2, $\text{slack}(C_2, s_0)$ is unchanged. In either case the hypotheses is true. $\blacksquare$

In Lemma 6, a result on commitment of a simple path common to two circuits is presented.

Lemma 6. Let C_1 and C_2 be two circuits intersecting along a subgraph G_1, where G_1 is not a closed path. Assume in s_0 a propagation occurs from C_1 to G_1. Then

$$\text{Comm}(G_1, s_0) < \text{Cap}(G_1)\text{-}1.$$

Proof. Since G_1 is not a closed path, the number of arcs in G_1 is less than $\text{Cap}(G_1)$-1; thus, in general,

$$\text{Comm}(G_1, s_0) \leq \text{Cap}(G_1) \text{-} 1 \quad \text{in any state } s \in S.$$

For propagation to occur from C_1 to G_1, there must be at least one uncommitted arc on G_1 in state s_0. Hence,

$$\text{Comm}(G_1, s_0) < \text{Cap}(G_1)\text{-}1. \qquad \blacksquare$$

Theorems 3 and 4 propose deadlock-free solutions to systems whose Wait Relation Graphs consist of two circuits. Theorem 3 considers two circuits intersecting in a single vertex, and Theorem 4 considers two circuits intersecting along a subgraph.

Theorem 3. Let Wait Relation Graph model G of a manufacturing system consist of two simple circuits, C_1 and C_2, which share a single vertex v. Figure 2.5 depicts a typical example of the system. The system is deadlock free if the following conditions are true:

1. $\text{slack}(C_1, s) > 0,$ (2-9a)
2. $\text{slack}(C_2, s) > 0,$ and (2-9b)

3. $\text{slack}(C_1 \cup C_2, s) > 1 \ , \forall s \in S.$ (2-9c)

Proof. Let the initial state s_0 satisfy (2-9). Define i and j such that

$$\text{slack}(C_i, s_0) \leq \text{slack}(C_j, s_0) \ .$$

Let s_1 be the state of the system after parts are propagated on C_i.

Case 1. Let

$$\text{slack}(C_i, s_0) = 1. \tag{2-10}$$

By application of Lemma 1 to circuit C_i, we know that $\text{slack}(C_i, s_1)$ cannot decrease. Therefore,

$$\text{slack}(C_i, s_1) \geq \text{slack}(C_i, s_0) > 0. \tag{2-11}$$

By Lemma 3

$$\text{slack}(C_i \cup C_j, s_0) = \text{slack}(C_i, s_0) + \text{slack}(C_j, s_0) - 1 > 1.$$

Using (2-10) this simplifies to

$$\text{slack}(C_j, s_0) > 1+1-1 = 1. \tag{2-12}$$

A part could propagate from C_i into C_j. But by Lemma 5 and (2-11) we obtain

$$\text{slack}(C_j, s_1) > 0. \tag{2-13}$$

Consider circuit $C_i \cup C_j$. By Lemma 1, slack($C_i \cup C_j$, s_0) cannot increase in subsequent states. Therefore,

$$\text{slack}(C_i \cup C_j, s_1) \geq \text{slack}(C_i \cup C_j, s_0) > 1. \tag{2-14}$$

From (2-11), (2-13) and (2-14), (2-9) is satisfied in state s_1, when slack(C_i, s_0) = 1.

Case 2. Let

$$\text{slack}(C_i, s_0) > 1. \tag{2-15}$$

From (2-9) and (2-15) we can conclude that in state s_0,

$$\text{slack}(C_i, s_0) > 1, \tag{2-16}$$

$$\text{slack}(C_j, s_0) > 1 \text{ and} \tag{2-17}$$

$$\text{slack}(C_i \cup C_j, s_0) > 1. \tag{2-18}$$

From Lemma 5

$$\text{slack}(C, s_1) \geq \text{slack}(C, s_0) \text{ -1} > 0$$

for circuits C_1 andC_2 . Lemma 1 shows that the slack($C_i \cup C_j$, s_0) cannot decrease. Thus Case 2 is proven.

The propagation can be repeated to remove all the parts from the system. Hence, the system is deadlock-free. ■

Theorem 4 considers two primary circuits intersecting along a subgraph. As the two circuits share vertices and arcs, the slack conditions required to keep the system deadlock-free differ from those in Theorem 3.

Theorem 4. Let the Wait Relation Graph model G of a manufacturing system consist of two simple circuits, C_1 and C_2, which intersect along a subgraph G_1, which contains more than one vertex. Figure 2.6 depicts a typical example of the system. Then, a propogation exists such that relation (2-19) holds before and after.

1. $\text{slack}(C_1, s) > 0$, (2-19a)
2. $\text{slack}(C_2, s) > 0$, and (2-19b)
3. $\text{slack}(C_1 \cup C_2, s) > 0, \ \forall s \in S.$ (2-19c)

Proof. Let the initial state s_0 satisfy (2-19). Define i and j such that

$$\text{slack}(C_i, s_0) \leq \text{slack}(C_j, s_0) \ .$$

Let s_1 be the state of the system after parts are propagated on C_i.

Case 1. Let

$$\text{slack}(C_i, s_0) = 1. \tag{2-20}$$

Applying Lemma 1 to circuit C_i, we know that $\text{slack}(C_i, s_1)$ cannot decrease. Therefore,

$$\text{slack}(C_i, s_1) \geq \text{slack}(C_i, s_0) > 0. \tag{2-21}$$

There are three different propagations on C_i which affect the slack on C_j:

a. Propagation occurs within G_1. Then

$$\text{slack}(C_j, s_1) = \text{slack}(C_j, s_0) > 0 \tag{2-22}$$

b. Propagation occurs from G_1 to C_i. Then

$$\text{slack}(C_j, s_1) = \text{slack}(C_j, s_0)+1> 0. \tag{2-23}$$

c. Propagation occurs from the C_i to G_1. From Lemma 4 and (2-20),

$$\text{slack}(C_i \cup C_j, s_o) = 1 + \text{slack}(C_j, s_o) - \text{Cap}(G_1) + \text{Comm}(G_1, s_o) > 1,$$

from which

$$\text{slack}(C_j, s_o) > \text{Cap}(G_1) - \text{Comm}(G_1, s_o) - 1 \tag{2-24}$$

follows. From Lemma 6 and (2-24), we obtain

$$\text{slack}(C_j, s_0) > 1. \tag{2-25}$$

From Lemma 5 and (2-25), we obtain

$$\text{slack}(C_j, s_1) > 0. \tag{2-26}$$

Consider circuit $C_i \cup C_j$. By Lemma 1, $\text{slack}(C_i \cup C_j, s_0)$ cannot increase in subsequent states. Therefore,

$$\text{slack}(C_i \cup C_j, s_1) \geq \text{slack}(C_i \cup C_j, s_0) > 0. \tag{2-27}$$

From (2-21), (2-22), (2-23), (2-26) and (2-27), slack conditions (2-19) are satisfied in state s_1, when $\text{slack}(C_i, s_0) = 1$.

Case 2. Let

$$\text{slack}(C_i, s_0) > 1. \tag{2-28}$$

From (2-28) and (2-19), we can conclude that in state s_0

$$\text{slack}(C_1, s_0) > 1, \tag{2-29}$$

$$\text{slack}(C_2, s_0) > 1 \text{ and} \tag{2-30}$$

$$\text{slack}(C_1 \cup C_2, s_0) > 1. \tag{2-31}$$

From Lemma 5,

$$\text{slack}(C, s_1) \geq \text{slack}(C, s_0) - 1$$

for all circuits C_1 andC_2 . Lemma 1 shows that the slack($C_i \cup C_j$, s_0) cannot decrease. Thus Case 2 is proven.

Therefore (2-19) holds for S_1. ■

The Wait Relation Graphs considered in Theorems 3 and 4 considered two circuits interacting in one or more vertices. The remaining theorems consider systems with Wait Relation Graphs consisting of two or more circuits interacting amongst each other in one or more vertices. In pursuing the deadlock-free solutions to these larger, less specific systems, the concept of order is introduced. As we will see, order is closely related to the deadlock free solutions of systems.

Order

The *Order* of a closed path is defined to be one less than the number of primary circuits which lie on the closed path and intersect any other primary circuit on the path at only one vertex. Order of a circuit C is abbreviated as order(C).

The order of a circuit is related to its structure. Order of a primary circuit is 0. The system considered in Figure 2.5 contained two circuits. C_1 and C_2 are of order 0, and $C_1 \cup C_2$ is of order 1. The system considered in Figure 2.6 contained three circuits--C_1, C_2, $C_1 \cup C_2$--all of order 0. Figure 2.7 depicts the Wait Relation Graph of a system. As an illustration of the definition of order, the order of some of the circuits in the graph is listed.

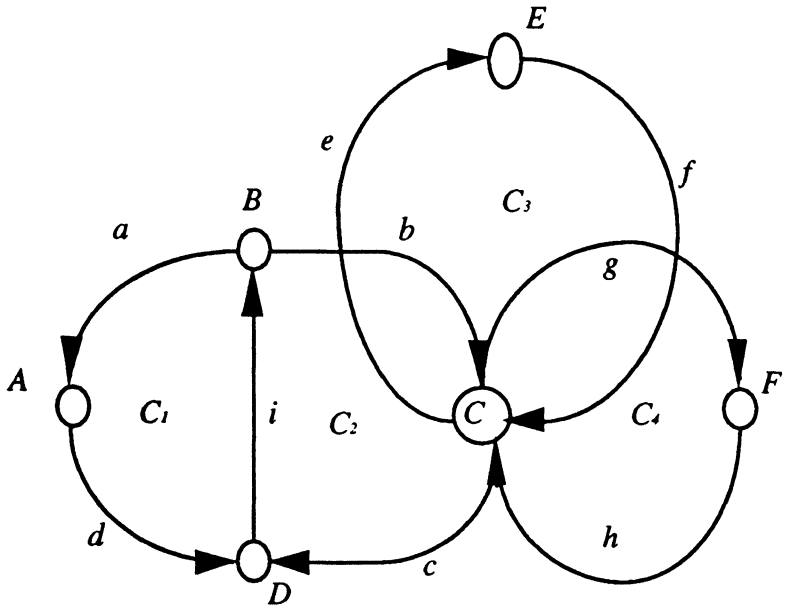

Figure 2.7. WRG of system in Example 2.5.1.

Example 2.5.1

The WRG in Figure 2.7 contains the circuit listed in Table 2.1.

C_1 = ({A, B, D}, {a, d, i})	order(C_1) = 0	
C_2 = ({B, D, C}, {b, c, i})	order(C_2) = 0	
C_3 = ({C, E}, {e, f})	order(C_3) = 0	
C_4 = ({C, F}, {g, h})	order(C_4) = 0	
$C_5 = C_1 \cup C_2$	order(C_5) = 0	C_1 and C_2 interesect in multiple vertices
$C_6 = C_4 \cup C_3$	order(C_6) = 1	C_4 and C_3 intersect in vertex C
$C_8 = C_2 \cup C_4$	order(C_8) = 1	C_2 and C_4 intersect in vertex C
$C_7 = C_2 \cup C_3$	order(C_7) = 1	C_2 and C_3 intersect in vertex C
$C_9 = C_1 \cup C_2 \cup C_4$	order(C_9) = 1	
$C_{10} = C_1 \cup C_2 \cup C_3$	order(C_{10}) = 1	
$C_{11} = C_2 \cup C_3 \cup C_4$	order(C_{11})= 2	
$C_{12} = C_1 \cup C_2 \cup C_3 \cup C_4$	order(C_{12}) = 2	

Table 2.1. Circuits in Figure 2.7.

Lemma 7.

Let C_1 and C_2 be two circuits intersecting each other. If C_1 and C_2 intersect in more than one vertice,

$$\text{order}(C_1 \cup C_2) = \text{order}(C_1) + \text{order}(C_2).$$

If C_1 and C_2 intersect in a single vertex,

$$\text{order}(C_1 \cup C_2) = \text{order}(C_1) + \text{order}(C_2) + 1.$$

Proof. The proof follows from the definition of order, and by observation in Example 2.5.1.

Theorem 5 considers the effect of altering the Wait Relation Graph of a deadlock-free system by the addition of a primary circuit, which intersects the graph at a single vertex. Conditions needed for deadlock-free operation of the new system are derived.

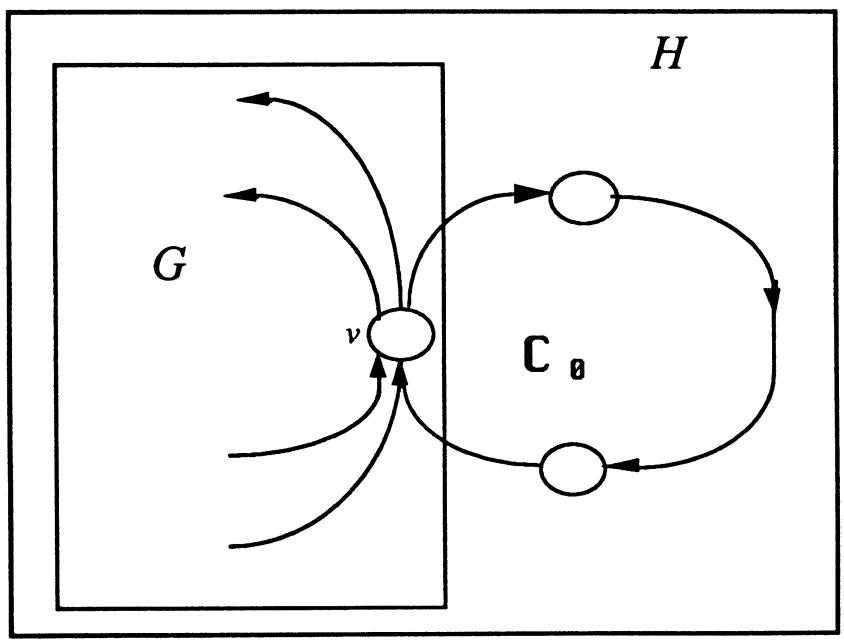

Figure 2.8. Circuit C_0 intersects graph G in a single vertex v.

Theorem 5. Let G and H be the WRG of two manufacturing systems with admissable states S_G and S_H. Assume that H is identical to G, except that H contains an additional circuit C_0 which is joined to G at vertex v. Suppose $\mathbf{C}_G$ is a set of circuits such that if

$$\text{slack}(C, s) > \text{order}(C) \; \forall \; C \in \mathbf{C}_G \text{ and } \forall \; s \in S_G, \tag{2-32}$$

then deadlock will not exist in the manufacturing system represented by G. If

$$\text{slack}(C, s) > \text{order}(C) \; \forall \; C \in \mathbf{C}_H \text{ and } \forall \; s \in S_H \tag{2-33}$$

where

$$\mathbf{C}_H = \mathbf{C}_G \cup \{C_0\} \cup \mathbf{C}_u, \tag{2-34}$$

and

$$\mathbf{C}_u = \{C: \; C = C_0 \cup C_i, \; C_i \in C_G, \text{ and } v \in C_i\}, \tag{2-35}$$

then deadlock will not exist in the manufacturing system represented by H. Figure 2.8 depicts the Wait Relation Graphs G and H.

Proof. Let the system represented by the WRG H be in state s_0. Define $C^* \in \mathbf{C}_H$ to be a circuit which minimizes the function

$$\text{space}(C, s_0) = \text{slack}(C, s_0) - \text{order}(C).$$

We will now show that if a part is propagated along circuit C^*, resulting in the new state s_1, then

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_H \tag{2-36}$$

Case 1. Let

$$\text{space}(C^*, s_0) = 1 \text{ and } C^* = C_0. \tag{2-37}$$

Combining Lemma 1 and (2-37) we obtain

$$\begin{aligned} \text{slack}(C_0, s_1) &\geq \text{slack}(C_0, s_0) = 1 \\ &> \text{order}(C_0) = 0 \end{aligned} \tag{2-38}$$

Since all circuits $C \in \mathbf{C}_u$ contain C_0, Lemma 1 can also be used to conclude

$$\text{slack}(C, s_1) \geq \text{slack}(C, s_0) > \text{order}(C) \; \forall \; C \in \mathbf{C}_u. \tag{2-39}$$

Define

$$\mathbf{C}_v = \{C \in \mathbf{C}_G : v \in C\} \tag{2-40}$$

as the set of all circuits in WRG G containing vertex v. Part propagation on C_0 will not affect any circuit which does not contain v. Hence,

$$\text{slack}(C, s_1) = \text{slack}(C, s_0) > \text{order}(C) \; \forall \; C \in \mathbf{C}_G - \mathbf{C}_v. \tag{2-41}$$

The definition expressed in (2-35) states that for each $C_i \in \mathbf{C}_v$ there exists a $C_j \in \mathbf{C}_u$, such that $C_j = C_i \cup C_0$. Combining Lemma 1 and the definition of order results in

$$\begin{aligned} \text{slack}(C_j, s_1) &\geq \text{slack}(C_j, s_0) \\ &> \text{order}(C_j) \\ &> \text{order}(C_i)+1. \end{aligned} \tag{2-42}$$

Combining Lemmas 3 and 5 with (2-42), (2-41) and (2-37) results in

$$\text{slack}(C_i, s_1) \geq \text{slack}(C_i, s_0) \text{ -1},$$

$$\geq \text{slack}(C_j, s_0) - \text{slack}(C_0, s_0) + 1 - 1,$$
$$> \text{order}(C_i) + 1 - \text{order}(C_0),$$
$$> \text{order}(C_i). \qquad (2\text{-}43)$$

Therefore, under the assumptions for Case 1, (2-36) holds for state s_1. This concludes Case 1.

Case 2. Let

$$\text{space}(C^*, s_0) = 1, \text{ and } C^* \in \mathbf{C}_H - \{C_0\}. \qquad (2\text{-}44)$$

To ensure that the propagation is possible--hence, implying that the system is deadlock free in state s_0--we must consider the effect of a propagation in C^* on every circuit $C \in \mathbf{C}_H$. To organize this task, the set $\mathbf{C}_H$ is divided into its constituent sets $\{C_0\}$, $\mathbf{C}_u$ and $\mathbf{C}_G$. A circuit C from each of these sets is then chosen and analysed to study the effect of the propagation over C^*. The proof for Case 2, therefore has three major subparts:

1. the circuit $C = C_0$ is considered,
2. a circuit $C \in \mathbf{C}_G$ is considered and
3. a circuit $C \in \mathbf{C}_u$ is considered.

Part 1.

A propagation over C^* can effect $\text{slack}(C_0, s_0)$ only if C^* intersects C_0 at vertex v. Using the definition of order from Lemma 3 and (2-44), we obtain

$$\text{slack}(C^* \cup C_0, s_0) = \text{slack}(C^*, s_0) + \text{slack}(C_0, s_0) - 1$$
$$= \text{order}(C^*)+1+\text{slack}(C_0, s_0)-1$$

$$= \text{order}(C^*)+\text{slack}(C_0, s_0). \tag{2-45}$$

We know from the definition of order that

$$\text{slack}(C^* \cup C_0, s_0) > \text{order}(C^* \cup C_0, s_0) = \text{order}(C^*) + 1. \tag{2-46}$$

Combining (2-45) and (2-46) we obtain

$$\text{slack}(C^* \cup C_0, s_0) = \text{order}(C^*)+\text{slack}(C_0, s_0) > \text{order}(C^*)+1,$$

from which

$$\text{slack}(C_0, s_0) > 1 \tag{2-47}$$

follows. From Lemma 5 and (2-47), we obtain the result

$$\text{slack}(C_0, s_1) > 0. \tag{2-48}$$

This concludes Part 1.

Part 2.

Consider $C \in \mathbf{C}_G$. Then both C^* and C belong to the set $\mathbf{C}_G$, but we know from (2-32) that

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_G. \tag{2-49}$$

Part 3.

Propagation on C^* can only affect $C \in \mathbf{C}_u$ if they intersect. Then there exists a $C_i \in C_G$,such that $C_i \cup C_0 = C$. From Part 2

$$\text{slack}(C_i, s_1) > \text{order}(C_i).$$

From Part 1

$$\text{slack}(C_0, s_1) > \text{order}(C_0).$$

Since C_i is contained in C, then

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_u. \tag{2-50}$$

From results in (2-48), (2-49) and (2-50)

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_H.$$

Therefore, under the assumptions for Case 2, (2-36) holds.

Case 3. $$\text{space}(C^*, s_0) > 1 \tag{2-51}$$

Hence,

$$\text{slack}(C, s_0) > \text{order}(C)+1 \; \forall \; C \in \mathbf{C}_H.$$

From Lemma 5, it is easy to see that the slack conditions (2-36) are all satisfied for s_1. Hence, all slack conditions are satisfied in state s_1, and Case 3 is proven.

The propagation can be repeated to remove all the parts from the system. Hence, the system is deadlock-free. ■

Theorem 6 considers the effect of altering the Wait Relation Graph of a deadlock-free system by the addition of a primary circuit which interesects the graph along a subgraph G_1 containing more than 1 vertex. Conditions needed for deadlock-free operation of the new system are derived. Figure 2.9 depicts a representation of the system.

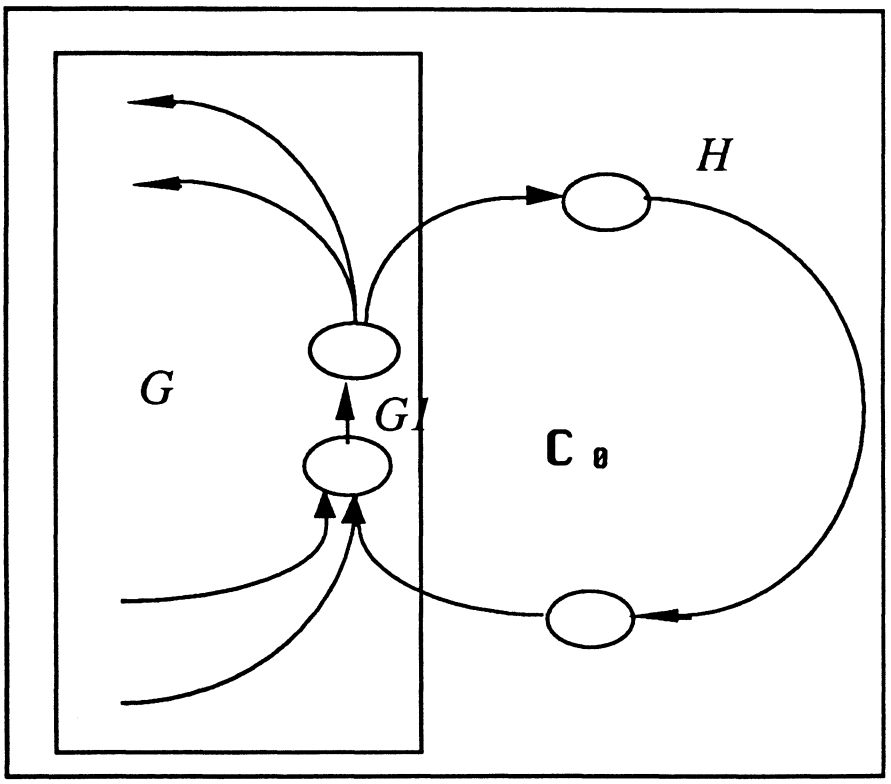

Figure 2.9. Circuit C_0 intersects graph G in simple path P.

Theorem 6. Let G and H be the WRG of two manufacturing systems with admissable states S_G and S_H. Assume that H is identical to G, except that H contains an additional circuit C_0 which is joined to G along a subgraph G_1, where G_1 contains more than 1 vertex. Suppose C_G is a set of circuits, such that if

$$\text{slack}(C, s) > \text{order}(C) \;\forall\; C \in \mathbf{C}_G \text{ and } \forall\; s \in S_G, \tag{2-52}$$

deadlock will not exist in the manufacturing system represented by G. If

$$\text{slack}(C, s) > \text{order}(C) \;\forall\; C \in \mathbf{C}_H \text{ and } \forall\; s \in S_H \tag{2-53}$$

where

$$\mathbf{C}_H = \mathbf{C}_G \cup \{C_0\} \cup \mathbf{C}_u, \tag{2-54}$$

and

$$\mathbf{C}_u = \{C:\ C = C_0 \cup C_i,\ C_i \in \mathbf{C}_G, \text{ and } C_i \text{ contains elements of } G_1\}, \tag{2-55}$$

then deadlock will not exist in the manufacturing system represented by H. Figure 2.9 depicts the Wait Relation Graphs G and H.

Proof. Let the system represented by the WRG H be in state s_0. Define $C^* \in \mathbf{C}_H$ to be a circuit which minimizes space(C, s_0). We will now show that if a part is propagated along circuit C^* resulting in the new state s_1, then

$$\text{slack}(C, s_1) > \text{order}(C) \;\forall\; C \in \mathbf{C}_H. \tag{2-56}$$

Case 1. Let

$$\text{space}(C^*, s_0) = 1 \text{ and } C^* = C_0. \tag{2-57}$$

Combining Lemma 1 and (2-57), we obtain

$$\begin{aligned} \text{slack}(C_0, s_1) &\geq \text{slack}(C_0, s_0) = 1 \\ &> \text{order}(C_0) = 0 \end{aligned} \tag{2-58}$$

Since all circuits $C \in \mathbf{C}_u$ contain C_0, Lemma 1 can also be used to conclude

$$\text{slack}(C, s_1) \geq \text{slack}(C, s_0) > \text{order}(C) \;, \forall\; C \in \mathbf{C}_u. \tag{2-59}$$

Define,

$$\mathbf{C}_P = \{C \in \mathbf{C}_G\text{: } C \text{ contains elements of } \mathrm{G}_1\}, \tag{2-60}$$

as the set of all circuits in WRG G containing at least a portion of G_1. Part propagation on C_0 will not affect any circuit which is not in $\mathbf{C}_P$. Hence,

$$\text{slack}(C, s_1) = \text{slack}(C, s_0) > \text{order}(C) \quad \forall\; C \in \mathbf{C}_G - \mathbf{C}_P. \tag{2-61}$$

The definition expressed in (2-55) states that for each $C_i \in \mathbf{C}_P$ there exists a $C_j \in \mathbf{C}_u$, such that $C_j = C_i \cup C_0$. Combining Lemma 1 and the definition of order results in

$$\text{slack}(C_j, s_1) \geq \text{slack}(C_j, s_0) > \text{order}(C_j) = \text{order}(C_i). \tag{2-62}$$

From Lemma 4 and (2-57),

$$\text{slack}(C_j, s_0) = \text{slack}(C_i, s_0) + \text{slack}(C_0, s_0) - \text{Cap}(\mathrm{G}_1) + \text{Comm}(\mathrm{G}_1, s_0) - 1 \tag{2-63}$$

Combining (2-63) and Lemma 5,

$$\text{slack}(C_i, s_1) \geq \text{slack}(C_j, s_0) - \text{slack}(C_0, s_0) + \text{Cap}(G_1) - \text{Comm}(G_1, s_0) + 1\text{-}1. \quad (2\text{-}64)$$

From (2-62), (2-64), Lemma 6 and substituting $\text{slack}(C_0, s_0) = 1$, we get

$$\text{slack}(C_i, s_1) > \text{order}(C_i). \quad (2\text{-}65)$$

From (2-58), (2-59), (2-61) and (2-65), (2-56) holds for state s_1.

Case 2. Let

$$\text{space}(C^*, s_0) = 1 \text{ and } C^* \in \mathbf{C}_H - \{C_0\}. \quad (2\text{-}66)$$

To ensure that the propagation is possible--implying that the system is deadlock-free in state s_0--we must consider the effect of a propagation in C^* on every circuit $C \in \mathbf{C}_H$. To organize this task, the set $\mathbf{C}_H$ is divided into its constituent sets $\{C_0\}$, $\mathbf{C}_u$ and $\mathbf{C}_G$. A circuit C from each of these sets is then chosen and analysed to study the effect of the propagation over C^*. The proof for Case 2, therefore, has three major subparts:

1. the circuit $C = C_0$ is considered,
2. a circuit $C \in \mathbf{C}_G$ is considered, and
3. a circuit $C \in \mathbf{C}_u$ is considered.

Part 1.

A propagation over C^* can effect $\text{slack}(C_0, s_0)$, only if C^* intersects C_0 in subgraph G_1. From Lemma 4 and (2-66), we obtain

$$\text{slack}(C^* \cup C_0, s_0) = \text{slack}(C^*, s_0) + \text{slack}(C_0, s_0) - \text{Cap}(G_1) + \text{Comm}(G_1, s_0)$$

$$= \text{order}(C^*)+1+\text{slack}(C_0, s_0) - \text{Cap}(G_1) + \text{Comm}(G_1, s_0). \quad (2\text{-}67)$$

We know from the definition of order that

$$\text{slack}(C^* \cup C_0, s_0) > \text{order}(C^*). \quad (2\text{-}68)$$

Combining (2-67) and (2-68), we obtain

$$\text{order}(C^*) +\text{slack}(C_0, s_0) +1 - \text{Cap}(G_1) + \text{Comm}(G_1, s_0) > \text{order}(C^*),$$

giving

$$\text{slack}(C_0, s_0) > \text{Cap}(G_1) - \text{Comm}(G_1, s_0) - 1,$$

which--using Lemma 6--simplifies to

$$\text{slack}(C_0, s_0) > 1. \quad (2\text{-}69)$$

From Lemma 5 and (2-69),

$$\text{slack}(C_0, s_1) \geq \text{slack}(C_0, s_0)-1,$$
$$> 0. \quad (2\text{-}70)$$

This concludes Part 1.

Part 2.

Consider $C \in \mathbf{C}_G$. Then both C^* and C belong to the set $\mathbf{C}_G$, but we know from (2-52) that

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_G. \tag{2-71}$$

Part 3.

Propagation on C^* can only affect $C \in \mathbf{C}_u$ if they intersect. Then there exists a $C_i \in C_G$, such that $C_i \cup C_0 = C$. From Part 2,

$$\text{slack}(C_i, s_1) > \text{order}(C_i).$$

Since C_i is contained in C, then

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_H. \tag{2-72}$$

From results in (2-70), (2-73) and (2-72)

$$\text{slack}(C, s_1) > \text{order}(C) \; \forall \; C \in \mathbf{C}_H.$$

Therefore, under the assumptions for Case 2, (2-56) holds.

Case 3. $$\text{space}(C^*, s_0) > 1. \tag{2-73}$$

Hence,

$$\text{slack}(C, s_0) > \text{order}(C)+1 \; \forall \; C \in \mathbf{C}_H.$$

From Lemma 5, it is easy to see that the slack conditions (2-56) are all satisfied for s_1. Hence, all slack conditions are satisfied in state s_1, and Case 3 is proven.

The propagation can be repeated to remove all the parts from the system. Hence, the system is deadlock-free. ■

The following example presents a method for obtaining all the slack conditions for a manufacturing system represented by a Wait Relation Graph. The Wait Relation Graph is broken up into primary circuits and paths. Any one simple circuit is chosen and a graph created. Paths and circuits are then added to the graph. The process of addition of paths and analysis of the resultant graph is repeated until the original graph is reached. At this stage, all circuits in the graph are known, and Theorems 5 and 6 can be used to determine all the slack conditions. An example of this process is given below.

Example 2.5.2

Consider a manufacturing system whose Wait Relation Graph is depicted in Figure 2.10.

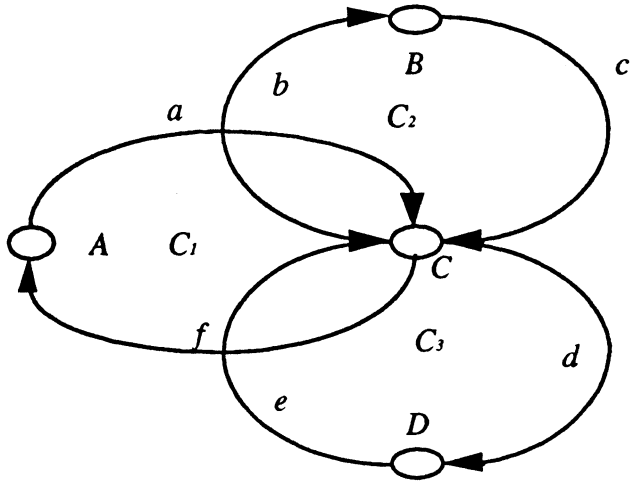

Figure 2.10. Wait Relation Graph of system in Example 2.5.2.

Consider a graph G^1 consisting of a single primary circuit C_1. Figure 2.11 depicts the situation.

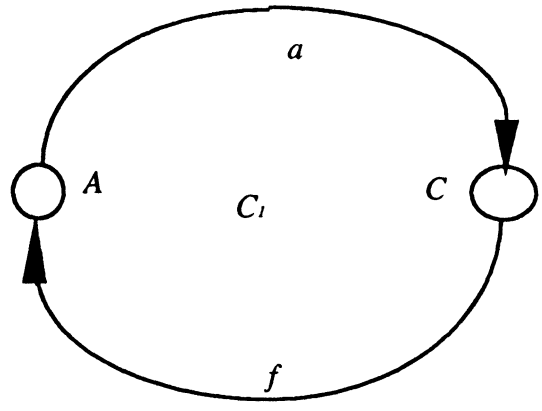

Figure 2.11. Wait Relation Graph G^1.

To G^1 add the path $P_1 = (\{B, C\}, \{b, c\})$.

Let G^2 be the resultant graph. Figure 2.12 depicts the situation.

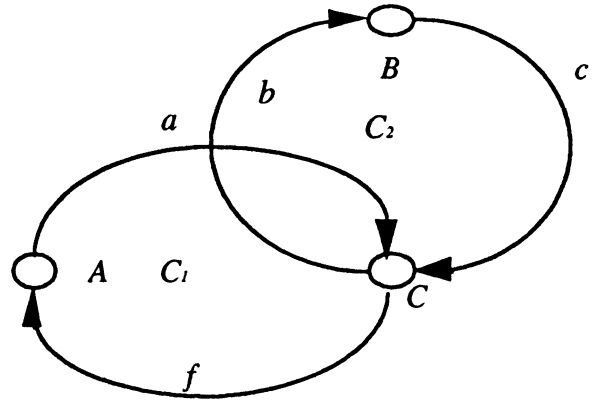

Figure 2.12. Wait Relation Graph G^2.

Let C^2 be the set of all circuits in G^2. The set $C^2 = \{ C_1, C_2, C_1 \cup C_2 \}$

Finally form graph $G^3 = G$ by adding path $P^2 = (\{C, D\}, \{d, e\})$ to G^2.

Let G^3 be the set of all circuits in G^3 . The set $C^3 = \{ C_1, C_2, C_3 , C_1 \cup C_2, C_2 \cup C_3, C_1 \cup C_3, C_1 \cup C_2 \cup C_3\}$.

The resulting slack conditions are:

1. slack(C_1, s) > 0;
2. slack(C_2, s) > 0;
3. slack(C_3, s) > 0;
4. slack($C_1 \cup C_2 , s$) > 1;
5. slack($C_1 \cup C_3 , s$) > 1;
6. slack($C_2 \cup C_3 , s$) > 1;
7. slack($C_1 \cup C_2 \cup C_3 , s$) > 2.

Theorem formally describes the method of Example 2.5.2.

Theorem 7. Let the Wait Relation Graph model of a manufacturing system consist of a graph G and a set of admissable states S_G. Then there exists a set of closed paths $\mathbf{C}_G$, such that G is deadlock-free if

$$\text{slack}(C, s) > \text{order}(C) \; \forall \; C \in \mathbf{C}_G, \forall \; s \in S_G. \tag{2-74}$$

Proof. Define the set $\boldsymbol{P} = \{P_i$: P_i is a path (open or closed) in G, only the first and last vertices in P_i may be common to any other path or circuit$\}$ and

$$\cup P_i = G$$

The result is proven inductively. Clearly $\boldsymbol{P}$ can be constructed to contain at least one circuit. Let C_1 be any circuit in $\boldsymbol{P}$. Form a graph G^1 consisting of C_1. By Theorem 2, G^1 is deadlock-free if

$$\text{slack}(C_i, s) > 0 = \text{order}(C_i). \tag{2-75}$$

Let $\mathbf{C}_{G_1} = C_1$; hence, the theorem holds for G^1. Assume the Graph G^i is deadlock-free, that is, $\text{slack}(C, s) > \text{order}(C)\ \forall\ C \in \mathbf{C}_{G^i}$. To form G^{i+1}, add a path P_i chosen from set $\boldsymbol{P}$ to G^i. The choice of P_i is not arbitrary; P_i must intersect G^i at both endpoints. There are two cases:

1. P_i intersects G^i in coincident vertices.
2. P_i intersects G^i at two vertices.

Case 1. Here a simple circuit C_0 is added to G^i at vertex v. Applying Theorem 5, G^{i+1} is deadlock-free if

$$\text{slack}(C, s) > \text{order}(C)\ \forall\ C \in \mathbf{C}_{G^{i+1}}\ \forall\ s \in S,$$

where $\mathbf{C}_{G^{i+1}}$ is the set of all circuits in G^{i+1}.

Case 2. Now, G^{i+1} is deadlock free if

$$\text{slack}(C, s) > \text{order}(C)\ \forall\ C \in \mathbf{C}_{G^{i+1}}\ \forall\ s \in S. \tag{2-76}$$

where $\mathbf{C}_{G^{i+1}}$ is the set of all circuits in G^{i+1}.

The proof follows by induction. ■

Theorem 7 presented a deadlock-free solution--one that could be applied to any manufacturing system. The important result was that a manufacturing system is deadlock-free, if

$$\text{slack}(C, s) > \text{order}(C)\ \forall C \in C_G \forall\ s \in S,$$

where $\mathbf{C}_G$ is constructed as in Theorem 7. Close examination of this construction theorem shows that $\mathbf{C}_G$ is also the set of all closed paths in the Wait Relation Graph G of the manufacturing system. Finding the deadlock free solution is a two-step process:

1. determining the set $\mathbf{C}_G$--that is all closed paths in G--and
2. applying (2-74) to determine the deadlock free slack conditions.

2.6 EXAMPLES

The theory developed in the previous sections is illustrated in two examples. In the first example, a small manufacturing system consisting of two processes and three resources is considered. Deadlock-free slack conditions are derived. The second example considers a larger system consisting of five resources and two processes. For each system, the Wait Relation Graph representation is formed and the deadlock-free slack conditions are derived.

2.6.1 EXAMPLE 1

Consider an example of a manufacturing system *M1* where two processes share two resources. The process plans are shown in Table 2.2.

Process No.	Process Plan
Process 1	Operation A processes Part 1 in Machine 1. Operation B transfers the part via Robot to Machine 2. Operation C processes Part 1 in Machine 2
Process 2	Operation D processes Part 2 in Machine 2. Operation E transfers the part via Robot to Machine 1. Operation F processes Part 2 in Machine 1

Table 2.2. The Process Plan for Example 1.

The Wait Relation Graph of the system is shown in Figure 2.13.

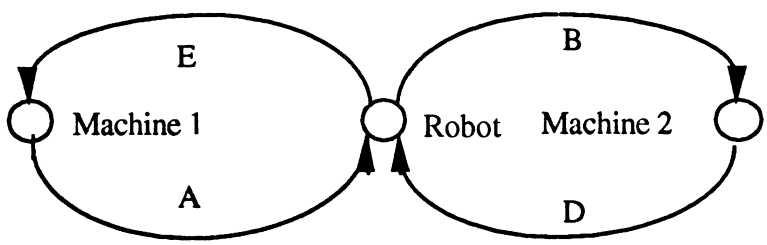

Figure 2.13. Wait Relation Graph representation of system M_1.

In M_1 the circuits are C_1, C_2, $C_1 \cup C_2$.

The deadlock-free slack conditions are:

1. slack(C_1, s) > 0;
2. slack(C_2, s) > 0; and
3. slack(C_3, s) > 1.

The deadlock-free slack conditions can be expressed as follows: only 1 operation from amongst A, B, C and D can be processing a part. Observance of this rule ensures that the system is deadlock-free.

2.6.2 EXAMPLE 2

In Example 2 a manufacturing system M_2, where two processes share five resources, is considered. The process plan is shown in Table 2.3

Process No.	Process Plan
Process 1	Operation A processes Part 1 in Machine 1. Operation B processes Part 1 in Machine 2. Operation C process Part 1 in Machine 3. Operation D processes Part 1 in Machine 4. Operation E processes Part 1 in Machine 5.
Process 2	Operation F processes Part 2 in Machine 5. Operation G processes Part 2 in Machine 3. Operation H process Part 2 in Machine 4. Operation I processes Part 2 in Machine 2. Operation J processes Part 2 in Machine 1.

Table 2.3. The Process Plan for Example 2.

The Wait Relation Graph of the system is shown in Figure 2.14.

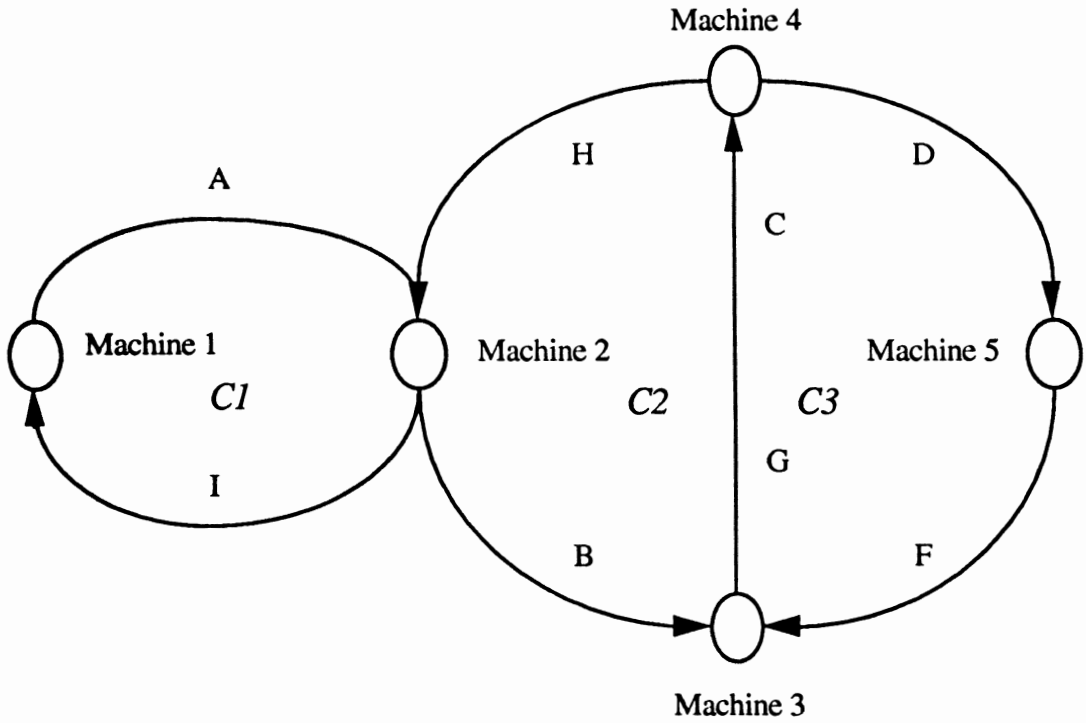

Figure 2.14: Wait Relation Graph representation of system M_2.

In Figure 2.14 there are three primary circuits--C_1, C_2, C_3. The rest of the closed paths are:

$C_4 = C_2 \cup C_3$;

$C_5 = C_1 \cup C_2$; and

$C_6 = C_1 \cup C_2 \cup C_3$.

The deadlock-free slack conditions are:

1. slack(C_1, s) > 0;
2. slack(C_2, s) > 0;
3. slack(C_3, s) > 0;
4. slack(C_4, s) > 0;

5. slack(C_5, s) > 1; and

6. slack(C_6, s) > 1.

The deadlock-free slack conditions can be expressed as a list of rules, which are to be observed always. These are:

1. Only 1 operation from amongst A and I can be processing a part.
2. Only 2 operation from amongst B, C, G and H can be processing a part.
3. Only 2 operation from amongst D, F, G and C can be processing a part.
4. Only 2 operations from amongst A, B, C, G, H and I can be processing a part.
5. Only 3 operations from amongst A, B, C, D, E, F, G, H and I can be processing a part.

Observance of these rules ensures that the system is deadlock-free.

2.7. COMPARISON OF PAST RESEARCH WITH METHOD IN THESIS

The deadlock detection and avoidance method developed in this thesis is more reliable and achieves better resource utilizations compared to methods in current research. For instance the Deadlock Detection Procedure(DDP)[1], does not detect all deadlock states. Consider the WRG in Figure 2.15. Here two processes share 5 resources. In the state shown, resource A is committed to operation O11 and resource E is committed to operation O21.

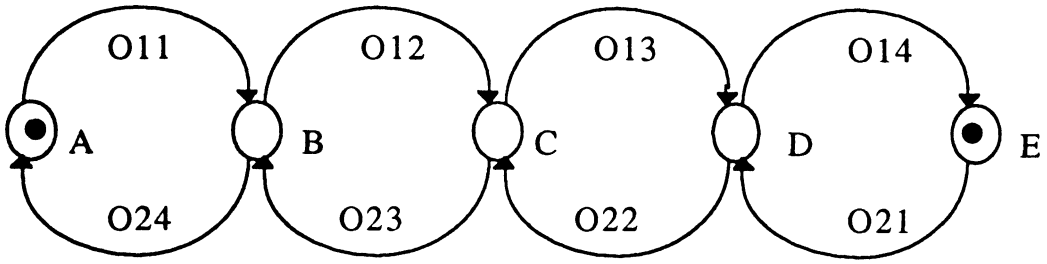

Figure 2.15. WRG of system with two processes and five resources.

The DDP would not identify this state as a deadlock state. However Theorem 7 can be used to prove it to be a circuit of order 3.

The method developed by Cho et al.[2] is also lacking in detection of higher level deadlocks. Their method uses buffers to break deadlock, which increase the number of system resources and at best postpone the occurrence of deadlock. No such buffer resources are used here. However, like Cho's method, some non-deadlocked states are incorrectly identified as deadlock. This affects resource utilizations but, more importantly, does not at all affect our goal, which is the detection of all deadlocks.

The deadlock avoidance algorithm in this thesis develops constraint conditions on groups of operations to be employed at process-runtime. One condition is developed for each of the deadlocks detected by the detection algorithm. The conditions allow for a maximum possible number of operations to be simultaneously active, while avoiding deadlock. The method has been tested and results compared with past research. Here we found that resource utilizations were higher than those in existing methods, such as PME's and DAA methods. Comparing the method of Zhou and DiCesare[3], we observe that PME's allocate resources to a process at the start of the process and/or release them in a group at the end of the process. Essentially, there exists some kind of allocation scheme which avoids free allocation of resources (resources allocated to an operation on demand and released immediately when the operation is done). However, this allocation scheme

will hold resources beyond their operation times, resulting in poor resource utilizations. No such allocation schemes are resorted to here, and resource utilizations are observed to be higher than PME methods.

For example consider a manufacturing system with three resources—a Mill, a Lathe and a Drill[8]. Assume three different parts are produced in accordance with the following process plans:

```
Part 1:    Mill,     Lathe.
Part 2:    Lathe,    Drill.
Part 3:    Drill,    Mill.
```

The Wait Relation Graph for the system is shown in Figure 2.16.

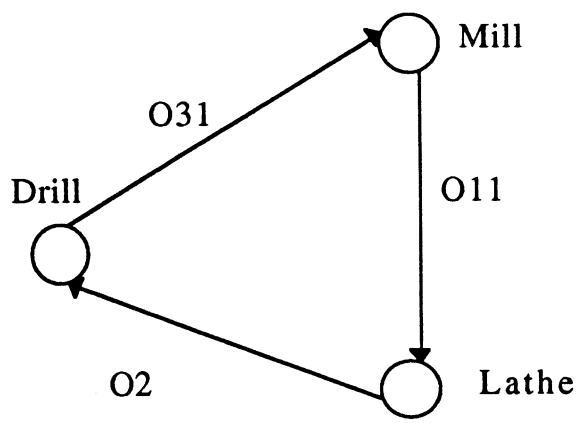

Figure 2.16. WRG of system with three processes and three resources.

Using Theorem 1, the system is deadlock free if slack(C, s) > 0 for all states in the system. There are however 7 states allowed by the slack condition which use all three resources. If the method developed by Zhou and Dicesare were used there would be 56 possible PME structures to model the shared resources. However every PME structure would prevent some of these 7 states from occurring.

Banaszak and Krogh's[4] DAA method requires that all shared resources needed by a part be allocated to the part at the outset of its entry to a resource zone. This holds resources longer than their operation times and results in poor utilizations. If DAA were used to prevent deadlock for the system in Figure 2.16, only one resource would be allowed into the system. Resources would idle for two-thirds of the operation time and utilizations would be low.

Hsieh and Chang [5] allow a greater number of resources to be processed concurrently than DAA, resulting in a higher resource utilization. However, no fixed guidelines on how the dispatching policy can be implemented are presented. This is especially true for their "Job Clearing Algorithm", where a choice of four procedures are presented. Their replacement procedure is search-based and may be unsuitable for real-time applications, especially in larger systems. The method in this thesis has achieved higher resource utilization than DAC.

CHAPTER 3
PROGRAM DEVELOPMENT - FORMATION OF THE WAIT RELATION GRAPH

Comparing the two examples in Section 2.6, it is apparent that the procedure for the second example is more involved than the first. There are three circuits in the Wait Relation Graph for the first example, compared to six for the second. Correspondingly the number of slack rules to be observed is also greater; three for the first example, compared to six for the second. It is clear that the deadlock analysis of any larger system would be quite involved and definitely a lengthy process. However, to determine the efficacy of the program, testing of a varied selection of systems is desirable. Also, one would like to test arbitrarily large systems--the ultimate test of the effectiveness of the theory. The process we have at hand is not adequate for these operations. A faster processing is required. We decided to write a computer program which would speed up this processing.

3.1 DESIGN ISSUES IN THE COMPUTER PROGRAM

In this section, we examine some of the design issues related to the program. As we are presently still in a concept-forming stage, the properties of the program are listed. There is no elaboration at this stage.

1. The computer program would be based on the theories in Chapter 2.
2. An input file would describe the manufacturing system.
3. The output would be a list of slack conditions.
4. The program should be able to process arbitrarily large systems. Here,

physical limitations of the operating system and platform will effect how large a system can be processed.

5. The final program will be in the C-language.
6. The program would be developed to run in the SUNOS 4.1 operating system.
7. The development process must be documented.

The development of the program will be described in the present and the next two chapters. Each chapter will describe one stage in the process: Chapter 3 will describe the formation of the Wait Relation Graph; Chapter 4 will describe the detection of all primary circuits; Chapter 5 will describe the detection of all higher-order circuits; and Chapter 6 will present examples of manufacturing systems to be analyzed using the program.

3.2. PROGRAM OVERVIEW

The program accepts an input file describing the system. The output is a list of slack conditions required to keep the system deadlock-free. The program itself is broken into three segments. Each of these segments performs a well-defined task.

The first segment models the system as a Wait Relation Graph. The program first reads in a description of the manufacturing system. This information is then processed, and the Wait Relation Graph representation of the system created. Development of the first segment is decribed in the present chapter.

The second segment utilizes the Wait Relation Graph, a string multiplication algorithm and a recursive algorithm to detect all primary circuits. A list storing all primary circuits is created. Development of this segment is described in Chapter 4.

The third segment utilizes the list of primary circuits to detect all circuits of orders greater than 0. Unions of primary circuits are formed and their orders determined. Each

such union--along with its order--is stored in a list of higher-order circuits. The slack conditions required to keep each circuit deadlock-free are obtained from the order of the circuit. Development of this segment is described in Chapter 5.

Development of each segment is done in stages. The first stage--algorithm development--is optional and reserved for those routines which are sufficiently involved to merit an initial algorithm design. In the next stage, important data structures are defined. Following that, pseudo code for all routines is developed. Finally the pseudo code is used to develop the program source code in the C-language. In each of the chapters, important data structures and pseudo code descriptions of routines are included. Algorithmic descriptions of important routines are included. The C-language source code is not listed.

3.3. SYSTEM DESCRIPTION IN TERMS OF INPUT FILE

The program learns of the system description through a file; henceforth called *input file*. A file description facilitates a conveninient means of describing the system to the program. The input file stores names of resources in the system and process plans. It consists of two distinct sections. The first section lists names of all resources used in the system. The second section lists the resource sequences for the different part types to be manufactured. Each resource sequence lists--in order--the names of the resources used by the operations. As each operation uses one resource, the number of resources in the process plan equals the number of operations. The exact format of the file is listed in Table 3.1.

FORMAT OF INPUT FILE	EXPLANATION OF INPUT FILE FORMAT (on a line-by-line basis).
RESOURCES	Header indicating that resources declaration follows
<r1> <r2> <r3>.. <rN>	List of resources in system.
PROCESS 1	Header indicating that operation sequence for Process 1 follows.
<rA> <rB> <rC>... <rI>	Operation sequence for process 1
PROCESS 2	Header indicating that operation sequence for Process 2 follows
<rP> <rQ> <rR>... <rJ>	Operation sequence for Process 2
...	...
PROCESS *M*	Header indicating that operation sequence for Process M follows
<rX> <rY> <rZ>.. <rK>	Operation sequence for Process *M*
END	End of file string. Indicates end of input file.

Table 3.1. Format of Input File.

Illustrative example

Consider once again the example of 'Dies Incorporated'. The input file for this system is given in Table 3.2.

```
RESOURCES
Robot Mill Lathe Drill
PROCESS 1
Robot Mill Robot Lathe Robot
PROCESS 2
Robot Lathe Robot
PROCESS 3
Robot Drill Robot
PROCESS 4
Robot Mill Robot Drill Robot
END
```

Table 3.2. Input File for Dies Incorporated.

3.4. DATA STRUCTURES

This section defines the major data structures that are used by the first segment of the program. They are defined using a C-type notation.

3.4.1. TRANS

This structure stores information about one operation in the system. Every operation is part of a process plan for some process *q*. The *operation_number* of a process

refers to the order of the operation within the operation sequence Oper(q). The *process_number* stores the value of q, the process number. A linked list of *trans* data structures stores information on all operations in the system. This data structure also stores the address of the next element in the linked list.

```
trans{
      short operation_number
      short process_number
      trans *next
      }
```

3.4.2. ARC_INFO

This structure stores information on the operations represented by each arc in the Wait Relation Graph of the system. The number of operations represented by the arc is stored. Detailed information on each of the operations is stored in a linked list of *trans* data structures. The memory address of the next element in the linked list is also included.

```
arc_info{
      short number_of_operations
      trans *next
      }
```

3.4.3. RESOURCE_DEF

This structure stores the resource name and resource number of a resource in the system. A linked list of *resource_def* structures is used to store information on all resources in the system. The address of the next element in the linked list is also included.

```
resource_def{
      short resource_number
      char *resource_name
      resource_def *next
      }
```

3.4.4. RESOURCES

This is a linked list of resource_def data structures. It stores information on all resources in the system. The address of the first element in linked list is stored.

```
resource_def *resources
```

3.4.5. WAIT GRAPH MATRIX

This array of *arc_info* data structures stores information on all arcs in the Wait Relation Graph of the system. The array is two-dimensional square. Size of the array is *N* x *N*, where *N* is the number of resources in the system. If an arc exists between vertices i and j in the Wait Relation Graph--the *ij*th element of the wait graph--then the array contains an arc info element; otherwise, it is NULL.

```
arc-info wait_graph_matrix[N] [N]
```

3.4.6. NUMBER OF RESOURCES

This global variable stores the number of resources in the system.

```
short number_of_resources
```

3.4.7. NUMBER OF PROCESSES

This global variable stores the number of processes in the system.

```
short number_of_processes
```

3.5 ROUTINES

In this section, pseudo code descriptions of all routines are included. The data structures described in previous sections are used in the pseudo code.

3.5.1. READ_RESOURCES_INFORMATION_FROM_INPUT_FILE

This routine extracts information on the resources in the system from the input file. The routine reads only the first two lines of the input file; the second lists the names of all resources in the system. Each of the resource names are extracted from the second line. Numbers are assigned to each resource, and this information is stored in a *resource_def* data structure. Each such structure is added to the *resources* linked list. The number of resources are counted and stored in the *number_of_resources* global variable. The routine returns a pointer to the *resources* linked list.

```
read_resources_information_from_input_file(input file)
start routine
  read second line from input file and store it in variable Line
  do{
    read next word in Line
    if (word is not NULL)
      increment number_of_resources
      store word and number_of_resources in a resource_def data
            structure
      add resource_def data structure to resources linked list
    end if
  while (word is not NULL)
  return  resources linked list
end routine
```

3.5.2. MAP_A_RESOURCE_NAME_TO_ITS_NUMBER

This routine accepts a name of a resource from the calling function. It then scans the resources linked list to isolate the resource number of the resource with the name. The resource number is returned to the calling function(Refer to section 3.4.3. for a declaration of the `resource_def` data structure.)

```
map_resource_name_to_number(name)
start routine
  for every element in resources linked list
    if ( name matches resource_name field of linked list element)
       return resource_number field of linked list element
    end if
  end for
  print error message and quit
end routine
```

3.5.3. INITIALIZE THE ARC ARRAY

This routine creates the Wait Graph matrix from the information stored in the input file. The input file is read one-line-at-a time starting from the third line. The line read could contain either a process name, an operation sequence for a process or the end of file string. The lines storing process names are skipped, but each line storing an Operation Sequence is further analysed. The names of resources in each Operation sequence are extracted. Two variables--*resource1*, *resource2* --are maintained in this process. The first stores the resource number of the present resource, and the second the resource number of the last resource extracted. A Wait Relation exists for every such pair of resources. Information on the operation is stored in the linked list of operations associated with the arc from *resource 1* to *resource 2* in the Wait Graph matrix. The process is repeated for every operation sequence. The routine stops when the end of file string (END string) is encountered. The number of processes in the system is counted by keeping account on the number of operation sequences analysed. The address of the newly created Wait Graph matrix is returned to the calling routine.

```
initialize_arc_array(input file)
start routine
  consider the second line in input file
  wait_graph = allocate (N x N units of arc_info, where N =
      Number_of_Resources)
  do{
    Line = next line in input file
    if ( Line does not contain "PROCESS" or is not equal to "END") then
       increment number_of_processes
       operation_number=0
      do{
         read next word in Line
         increment operation_number
         if (word read is first in Line) then
            map word to its corresponding resource number in the
                  resources list by calling the
            map_resource_name_to_number function.  Store this in
      variable resource2
         else
            resource1 = resource2
            map word to its corresponding resource number in the
                  resources list by calling the
            map_resource_name_to_number function.  Store this in
      variable resource2
            ptr = allocate(1 unit of trans)
            ptr.process_number = number_of_processes
            ptr.operation_number = operation_number
            ptr.next = wait_graph[resource1][resource2].trans
            wait_graph[resource1][resource2].trans = ptr
         end if
      while ( word is not NULL)
    end if
  while ( Line is not END)
  close input_file
```

```
  return wait_graph
end routine
```

CHAPTER 4
PROGRAM DEVELOPMENT - PRIMARY CIRCUIT DETECTION

The present chapter explains the development of the second segment of the program; namely, the detection of all primary circuits in the Wait Relation Graph. The chapter is divided into two parts. In the first part, the theory of string multiplication, matrix multiplication and circuit extraction is described. Section 4.1 explains the theory of string multiplication, and Section 4.2 explains circuit extraction. In the second part of the chapter, the implementation of the theory is explained. Section 4.3 consists of data structure declarations. Section 4.4 contains the pseudo code descriptions of the routines.

4.1 STRING MULTIPLICATION THEORY

A string multiplication algorithm contained in the paper, "Detection of deadlocks in Flexible Manufacturing Cells", by Wysk, Joshi and Yang [1], is used to identify all primary circuits in the Wait Relation Graph. First, a symbol matrix **S** is defined from the Wait Relation Graph *G* of the system.

Definition: Symbol matrix **S** is a matrix of order N x N, where

$s_{ij} = ij$, if an arc exists between vertices i and j in the Wait Relation Graph; otherwise,

$s_{ij} = 0$

and N is the Number of Resources in the system.

The string multiplication technique applies to any two strings of symbols. Let uv and vw be two strings of symbols that start and end with v, respectively. Let * denote the string multiplication symbol. Then,

$$u * 0 = 0 * u = 0. \tag{4-1}$$

The product of the strings uv and vw is formed by concatenating uv with the string that results from vw by removing the first symbol v in vw. Hence,

$$uv * vw = uvw. \tag{4-2}$$

The result can be extended to sums of strings by defining,

$$\sum_i u_i v * \sum_j v w_j = \sum_i \sum_j u_i v * v w_j \ . \tag{4-3}$$

For example,

$$(av + bv) * (vc + vd) = av * vc + av * vd + bv * vc + bv * vd,$$
$$= avc+avd+bvc+bvd.$$

The next issue deals with matrix multiplication involving symbol matrix **S** and powers of **S**. The string multiplication technique described is used to form the product of individual strings. The product of **S** with itself is defined as

$$[\mathbf{S}^2]_{ij} = \sum_{k=1}^{N} s_{ik} * s_{kj} \tag{4-4}$$

where $[\mathbf{S}]_{ij}$ denotes the ij element of the matrix enclosed in the brackets.

Example 4.1. Let **S** be the symbol matrix for a Wait Relation Graph *G* of the system represented in Figure 4.1.

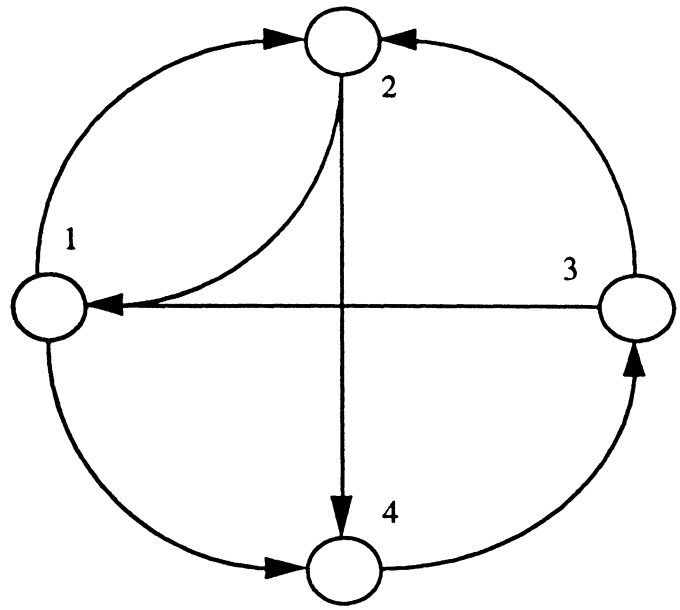

Figure 4.1. Wait Relation Graph *G* of System in Example 4.1.

$$\mathbf{S} = \begin{bmatrix} 0 & 12 & 0 & 14 \\ 21 & 0 & 0 & 24 \\ 31 & 32 & 0 & 0 \\ 0 & 0 & 43 & 0 \end{bmatrix}$$

Using (4-4) the matrix $\mathbf{S}^2$ is obtained:

$$\mathbf{S}^2 = \begin{bmatrix} 121 & 0 & 143 & 124 \\ 0 & 212 & 243 & 214 \\ 321 & 312 & 0 & 314+324 \\ 431 & 432 & 0 & 0 \end{bmatrix}$$

The matrix $\mathbf{S}^2$ contains redundant diagonal strings. For example, $[\mathbf{S}^2]_{11} = [\mathbf{S}^2]_{22}$. These redundancies arise, because a circuit can be expressed in various equivalent forms. Consider a circuit between three nodes--1, 2 and 3. This circuit could be expressed as 1231, 2312 or 3123. One way of eliminating these redundancies is to choose the string with the lowest starting index--in this case 1231--and eliminate the rest.

In order to eleminate redundancies in the manner described, the formulation used in calculating $\mathbf{S}^n$ is changed as follows:

$$\mathbf{S}^n = \mathbf{S}^{n-1\Delta} * \mathbf{S} \ , \tag{4-5}$$

where

$\mathbf{S}^{\Delta}$ is the upper triangular matrix of **S**.

As a result of (4-5), the numerical value of the first symbol in any string will always be less than that of the remaining symbols. Hence duplications are avoided within the symbol matrix.

Equation (4-5) can be expressed in a form more suitable to computation, as follows:

$$[\mathbf{S}^n]_{ij} = \sum_{k=i+1}^{N} [\mathbf{S}]_{ik} * [\mathbf{S}^{n-1}]_{kj}. \qquad (4\text{-}6)$$

There are three other steps that can be taken to avoid calculating unnecessary paths:

1. To eliminate calculating paths that circle the same circuit multiple times, the diagonal elements are removed from $\mathbf{S}^m$ before calculating $\mathbf{S}^{m+1}$.
2. Since the first element of a circuit must be less than all the other elements, then circuits starting with 1 can contain all N nodes, but circuits starting with node m can contain at most N-m+1 nodes. (Recall we are trying to find all the primary circuits; therefore, a node can appear at most once.) Because of this observation, there is no need to calculate the last m-1 diagonal elements of $\mathbf{S}^m$.
3. Finally, since the first element of every circuit must be less than all the other nodes, there is no reason to retain any path that has a node less than the first node. Therefore, in row m of $\mathbf{S}^m$ the off-diagonal paths cannot be any longer than N-m nodes long. Thus, in $\mathbf{S}^m$ none of the off-diagonal elements in the last N-m rows need to be calculated.

Example 4.2. Using (4-6) and symbol matrix **S** as defined in Example 4.1, matrix $\mathbf{S}^2$ is recalculated as

$$\mathbf{S}^2 = \begin{bmatrix} 121 & 0 & 143 & 124 \\ 0 & 0 & 243 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

It is interesting to note that the element $[\mathbf{S}^2]_{ij}$ is comprised of strings which represent paths with two arcs between nodes i and j. The diagonal elements $[\mathbf{S}^2]_{ii}$ represent all closed paths with *two* arcs which include node i. Hence two arc circuits are in the diagonal elements of $\mathbf{S}^2$, three arc circuits are in the diagonal elements of $\mathbf{S}^3$ and so on, as described below.

For $\mathbf{S}^3$

$$\mathbf{S}^3 = \begin{bmatrix} 1431 & 1432 & 1243 & 0 \\ 2431 & 2432 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

For $\mathbf{S}^4$

$$\mathbf{S}^4 = \begin{bmatrix} 14321+\ 12431 & 12432 & 0 & 14324 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

4.2 IMPLEMENTATION ISSUES

In this section, implementation issues which arise in the identification of circuits using the string multiplication algorithm in Section 4.1 are examined.

In the symbol matrix, each symbol in a string represents a node in the Wait Relation Graph. The string *ijk* represents a path from vertex i to j to k. The present implementation

is fine provided there are enough unique symbols to represent nodes. In the case of large systems, this issue can be a problem. Consider a graph comprising 100 nodes. If the nodes are numbered from 1 to 100, a string 123 could represent a path from node 1 to 2 to 3 or a path from node 12 to 3 or a path from node 1 to 23. Alphabetical symbols assigned to represent nodes do not work any better. There is a finite set of such alphabets--and no matter how many--these are inadequate to represent a general system. One solution is to use parantheses. The path from 1 to 2 to 3 is represented as 1(2(3)). The next section develops this idea and presents a modified string multiplication algorithm.

4.2.1. RULES FOR STRING MULTIPLICATION

Let $u(v)$ and $v(w)$ be two strings of symbols that end and start with symbol v, respectively. Symbols u and$w$ can themselves be single symbols or strings of symbols. Symbol v, is however, a single symbol. Let * denote the string multiplication symbol. Then

$$u(v) * 0 = 0 * u(v) = 0. \tag{4-7}$$

The product of $u(v)$ and $v(w)$, $u(v) * v(w)$ is formed in these steps:

1. Strip the trailing parantheses from $u(v)$, to give $u(v$.
2. Strip the leading symbol from $v(w)$ to give (w).
3. Concatenate the $u(v$ with (w) to give $u(v(w)$.
4. Add a trailing paranthesis to $u(v(w)$ to give the final product string, $u(v(w))$.

Steps 1 to 4 can be summarized by the equation

$$u(v) * v(w) = u(v(w)). \tag{4-8}$$

Equation (4-8) can be extended to

1. products of sums of strings, as

$$\sum_i u_i(v) \; * \sum_j v(w_j) = \sum_i \sum_j \; u_i(v(w_j)) \text{ and} \tag{4-9}$$

2. sums of products of strings, as

$$\sum_i v(u_i) \; * \; u_i(w) = v(\sum_i u_i\,(w)). \tag{4-10}$$

Example 4.3. The product of the two sums of strings,

$$(a(v) + b(v)) * (v(c) + v(d))$$

is formed using (4-9).

$$\begin{aligned}(a(v) + b(v)) * (v(c) + v(d)) = \; & a(v) * v(c) + a(v) * v(d) + b(v) * v(c) + b(v) * v(d) \\ & = a(v(c)) + a(v(d)) + b(v(c)) + b(v(d)).\end{aligned}$$

Example 4.4. The sum of the two products of strings,

$$a(v) * v(d) + a(w) * w(d),$$

is formed using (4-10).

$$a(v) * v(b) + a(w) * w(d) = a((v+w)d).$$

4.2.2. REDEFINITION OF THE S MATRIX

In order to use equations (3-7), (4-8), (4-9) and (4-10), the elements of the **S** matrix should be in a form required by these equations. The **S** matrix is now formed as

s_{ij} = (*ij*) if an arc exists between nodes *i* and *j* in the Wait Relation Graph

otherwise.

The products of **S** are formed using (4-6), where string multiplication is defined as in (4-7), (4-8), (4-9) and (4-10).

Example 4.5. The symbol matrix in Example 4.1 is formed using the representation presented in this section. Hence,

$$\mathbf{S} = \begin{bmatrix} 0 & 1(2) & 0 & 1(4) \\ 2(1) & 0 & 0 & 2(4) \\ 3(1) & 3(2) & 0 & 0 \\ 0 & 0 & 4(3) & 0 \end{bmatrix}$$

Using equations (4-6) to (4-10), the matrix $\mathbf{S}^2$ is obtained as

$$\mathbf{S}^2 = \begin{bmatrix} 1(2(1)) & 0 & 1(4(3)) & 1(2(4)) \\ 0 & 0 & 2(4(3)) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In Example 4.5, primary circuits with two arcs occur along the diagonal of matrix $\mathbf{S}^2$. The circuit from node 1to node 2 to node 1 is an example. The matrices $\mathbf{S}^3$ and $\mathbf{S}^4$ are also evaluated.

$$\mathbf{S}^3 = \begin{bmatrix} 1(4(3(1))) & 1(4(3(2))) & 1(2(4(3))) & 0 \\ 2(4(3(1))) & 2(4(3(2))) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

$$\mathbf{S}^4 = \begin{bmatrix} 1((4(3(2))+2(4(3)))1) & 1(2(4(3(2)))) & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Finally, all the circuits are,

1 2 1, 1 4 3 1, 2 4 3 2, 1 4 3 2 1 and 1 2 4 3 1.

Example 4.6. Let **S** be the symbol matrix of a Wait Relation Graph G depicted in Figure 4.2.

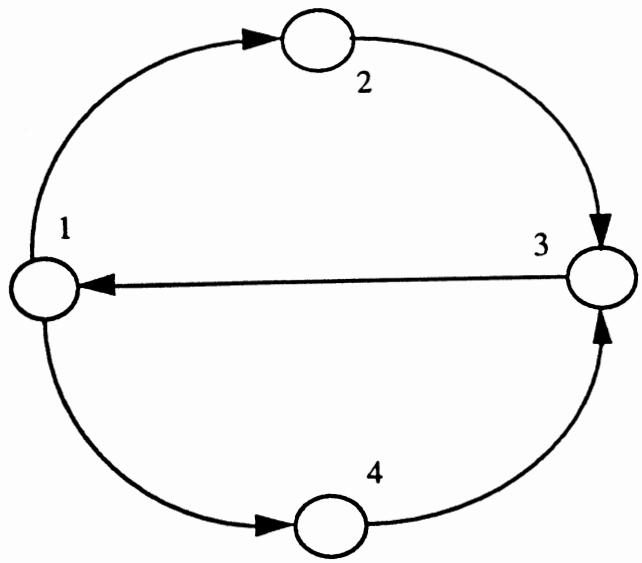

Figure 4.2. The Wait Relation Graph of the system in Example 4.6

$$\mathbf{S} = \begin{bmatrix} 0 & 1(2) & 0 & 1(4) \\ 0 & 0 & 2(3) & 0 \\ 3(1) & 0 & 0 & 0 \\ 0 & 0 & 4(3) & 0 \end{bmatrix}$$

Using equations (4-6) to (4-10),

$$\mathbf{S}^2 = \begin{bmatrix} 0 & 0 & 1((2+4)3) & 1(2(4)) \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

and

$$\mathbf{S}^3 = \begin{bmatrix} 1(((2+4)3)1) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

In Example 4.6, primary circuits with 3 arcs occur along the diagonal elements of matrix $\mathbf{S}^3$. Hence, the element $[\mathbf{S}^3]_{11} = 1(((2+4)3)1)$ represents a circuit string, although this is not immediately apparent. Recall that the motivation for the current representation was to unambiguously represent a string. Although this issue is solved, a new--issue,namely that of extracting the circuit string--remains to be solved.

4.2.3. THE EXTRACTION OF ALL CIRCUITS

From Example 4.6 consider once again the string $[\mathbf{S}^3]_{11} = 1(((2+4)3)1)$. Being a diagonal element, this represents a circuit with three arcs, including node 1. The two primary circuits are 1(2(3(1))) and 1(4(3(1))). This information is in the string and needs to be extracted. In this section, an algorithm which extracts primary circuits from each diagonal element is explained. The algorithm is recursive in nature.

4.2.3.1. DEFINITIONS

Before explaining the algorithms, a few concepts needed by the routine are described.

String pointer

A *string pointer* is an operator associated with a symbol string. Its value indicates the position in the symbol string at which the next character to be read is located. For any string, the string pointer has its lowest value when it refers to the first character in the string, and its highest value at the last character in the string. Initially, the string pointer has its lowest value. Incrementing the string pointer moves it one position to the right in the string.

Level

Level is an attribute of a string pointer. Level is assigned a value equal to the number of '(' symbols less the number of ')' symbols which are present below the current location of the string pointer. A string pointer is said to advance a level when it moves past a `(' symbol. It decrements a level when it moves past a ')' symbol. The level attribute is very useful in extracting circuits when there are multiple circuits present in the symbol string. It is also useful as an end-of-string indicator.

A note on representation

In all algorithms, henceforth, circuits will be stored as a sequence of node numbers separated by spaces. The nodes are arranged in the string in the order in which they occur in the circuit. The circuit string " a b c ...k a " represents a circuit from node a to b to c and so on, finally including node k and back to node a. This circuit would have been represented in the earlier representation as "a(b(c(...(k(a))...)))".

4.2.3.2. ALGORITHM

This section introduces the algorithm used to extract all primary circuits from a symbol string. The algorithm is recursive.

INPUTS TO ALGORITHM

a. A symbol string in the form obtained from the string multiplication routine; e.g. 1(((2+4)3)1).

b. The location of the string pointer--initially at the start of the symbol string.

c. The current list of circuits extracted from the symbol string, henceforth called *circuit_list*. As new circuits are extracted they are added to this list.

d. The current value of the level attribute.

INITIAL SETUP

Move the string pointer along the symbol string updating the level attribute at each step. Stop at the first numeric character in the symbol string.

Create a blank string, and store the numeric symbol in it. This string is, henceforth, called *circuit_string*.

Call the recursive algorithm. To it pass the symbol string, current string pointer location, current circuit_list and current value of the level attribute.

RECURSIVE ALGORITHM

Position the string pointer at its location within the symbol string.

Move the string pointer to the next symbol in the input string.

Based on the nature of the current symbol, various actions are performed:

If symbol is '(':

Increment level attribute.
Call the recursive algorithm with the current values of string ptr, level, circuit_string, circuit_list and the symbol string.

If symbol is ')':
Decrement the level attribute.
Return from algorithm.

If symbol is a numeric character:
Add the symbol to circuit_string.

If symbol is '+':
Store values of circuit string, string_pointer and level in temporary variables.
Move the string pointer along the symbol string updating level attribute at every step.
Stop at the first ')' symbol at which the level attribute is one less than the earlier stored value of level.
Call the recursive algorithm with appropriate parameters.
On returning from the algorithm, restore the earlier stored values to level, string_pointer and circuit_string.

If symbol is the end of string marker:
The string pointer has reached the end of the string. Return from the algorithm

If level attains a value of zero, it implies that the string pointer has reached the last `)` in the string. This automatically implies that a circuit has been extracted from the symbol string. Now add the string stored in circuit_string to circuit_list.

OUTPUTS FROM ALGORITHM

When the final circuit has been extracted, circuit_list will contain a list of all circuits extracted from the symbol string. These circuits will be stored in the form described above (Section 4.2.3.1); e.g., 1 2 3 1 and 1 4 3 1.

The next section contains an example based on the algorithm.

4.2.3.3. ILLUSTRATIVE EXAMPLE

The method used to extract all circuits will be illustrated with the detailed explanation of the extraction of all circuits from a symbol string.

Consider the following example of a typical symbol string:

```
1((4(3(2))+2(4(3)))1)
^
```

The position of the string pointer is indicated by a '^'. A few variables which help us track our location in the recursions are also reported. The first of these is *recursion number*. It is incremented when the recursive algorithm is called, decremented on return from the recursive algorithm. The current value of level is also reported.

1. Current symbol is '1'. Store it in circuit_string C_1 = " 1". Level is 0, recursion number is 1. Call the recursive algorithm.

   ```
   1((4(3(2))+2(4(3)))1)
   ^
   ```

2. Move the string pointer to the next character. Current symbol is '('. Level is 1, recursion number is 1. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
 ^
```

3. Move the string pointer is moved to the next character. Current symbol is '('. Level is 2, recursion number is 2. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
  ^
```

4. Move the string pointer to the next character. Current symbol is '4'. It is added to circuit_string C_1 = " 1 4". Level is 2, recursion number is 3.

```
1((4(3(2))+2(4(3)))1)
   ^
```

5. Move the string pointer to the next character. Current symbol is '('. Level is 3, recursion number is 3. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
    ^
```

6. Move the string pointer to the next character. Current symbol is '3'. It is added to circuit_string to give C_1 = " 1 4 3". Level is 3, recursion number is 4.

```
1((4(3(2))+2(4(3)))1)
     ^
```

7. Move the string pointer to the next character. Current symbol is '('. Level is 4, recursion number is 4. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
      ^
```

8. Move the string pointer to the next character. Current symbol is '2'. It is added to circuit_string to give C_1 = " 1 4 3 2". Level is 4, recursion number is 5.

```
1((4(3(2))+2(4(3)))1)
        ^
```

9. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 3, recursion number is decremented to 4. Return from the algorithm.

```
1((4(3(2))+2(4(3)))1)
          ^
```

10. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 2, recursion number is decremented to 3. Return from the algorithm.

```
1((4(3(2))+2(4(3)))1)
           ^
```

11. Move the string pointer to the next character. Current symbol is '+'. Store the present values of circuit_string, level, and string_pointer in temporary variables *temp*1, *temp*2, *temp*3, respectively. Level is 2, recursion number is 3.

```
1((4(3(2))+2(4(3)))1)
            ^
```

12. Move the string pointer forward updating level at each step. Stop at the ')' symbol at which level is one less than the stored value *temp*2. Level is 1, recursion number 3.

```
1((4(3(2))+2(4(3)))1)
                  ^
```

13. Call the recursive algorithm. Level is 1, recursion number is 4.

14. Move the string pointer to the next character. Current symbol is '1'. It is added to circuit_string to giveC_1 = " 1 4 3 2 1". Level is 1, recursion number is 4.

```
1((4(3(2))+2(4(3)))1)
                  ^
```

15. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 0, recursion number is decremented to 3. Return from the algorithm. Final circuit string C_1 = " 1 4 3 2 1".

```
1((4(3(2))+2(4(3)))1)
                  ^
```

16. The program execution now returns to recursion 3. Earlier stored values of string_pointer (*temp*3) and circuit_string(*temp*1) are restored. The string pointer is now at symbol '+'. This indicates a multiple circuit exists. Now create a new circuit string C_2 and add to it the symbols in circuit_string. Hence, C_2 = " 1".

```
1((4(3(2))+2(4(3)))1)
        ^
```

17. Move the string pointer to the next character. Current symbol is '2'. It is added to circuit_string C_2 = " 1 2". Level is 2, recursion number is 3.

```
1((4(3(2))+2(4(3)))1)
          ^
```

18. Move the string pointer to the next character. Current symbol is '('. Level is 3, recursion number is 3. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
          ^
```

19. Move the string pointer to the next character. Current symbol is '4'. It is added to circuit_string to give C_2 = " 1 2 4". Level is 3, recursion number is 4.

```
1((4(3(2))+2(4(3)))1)
            ^
```

20. Move the string pointer to the next character. Current symbol is '('. Level is 4, recursion number is 4. Call the recursive algorithm.

```
1((4(3(2))+2(4(3)))1)
                ^
```

21. Move the string pointer to the next character. Current symbol is '3'. It is added to circuit_string to give C_2 = " 1 2 4 3". Level is 4, recursion number is 5.

```
1((4(3(2))+2(4(3)))1)
                ^
```

22. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 3, recursion number is decremented to 4. Return from the algorithm.

```
1((4(3(2))+2(4(3)))1)
                ^
```

23. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 2, recursion number is decremented to 3. Return from the algorithm.

```
1((4(3(2))+2(4(3)))1)
                ^
```

22. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 1, recursion number is decremented to 2. Return from the algorithm.

```
1((4(3(2))+2(4(3)))1)
                ^
```

23. Move the string pointer to the next character. Current symbol is '1'. It is added to circuit_string to giveC_2 = " 1 2 4 3 1". Level is 1, recursion number is 2.

```
1((4(3(2))+2(4(3)))1)
                ^
```

24. Move the string pointer to the next character. Current symbol is ')'. Level is decremented to 0, recursion number is decremented to 1. Return from the

algorithm. Final circuit string C_1 = " 1 2 4 3 1".

```
1((4(3(2))+2(4(3)))1)
                ^
```

25. Move the string pointer to the next character. Current symbol is '\0'. Level is 0, recursion number is decremented to 0. Return from the recursive algorithm to the parent routine.

```
1((4(3(2))+2(4(3)))1)
                ^
```

26. The final circuits are C_1 = " 1 4 3 2 1", and C_2 = " 1 2 4 3 1". The process is complete.

Sections 4.1 and 4.2 explained the theory and implementation issues in the extraction of all primary circuits from the Symbol matrix. The next section contains pseudo code descriptions for the routines which implement the extraction. The routines are based on the string multiplication and circuit extraction algorithms described in the past two sections.

4.3. CONSTANTS AND DATA STRUCTURES

This section declares constants and data structures used in the program

4.3.1. CONSTANTS

The following constants are used in this section of the program.

NEXT_TERM = 0

END_EXPRESSION = 1

4.3.2. DATA STRUCTURES

This section defines the major data structures that are used by the routine. They are defined using a C type notation.

4.3.2.1. S_ELEMENT

This structure stores information on paths between a pair of nodes in the Wait Relation Graph. The *source* and *sink* vertex numbers are stored. The directed arc present between these two vertices is also stored in *initial_string*. In case this arc is absent, the null string is stored in its place. Recall the s-matrix multiplication formula, $\mathbf{S}^{i} = \mathbf{S}^{\Delta *} \mathbf{S}^{i-1}$. The s-element structure stores those symbol strings in the $\mathbf{S}^{i-1}$ and $\mathbf{S}^{i}$ matrix, which represent paths between the source and sink vertex. The symbol string in the $\mathbf{S}^{i-1}$ matrix is stored in *string2*, and the symbol string in the $\mathbf{S}^{i}$ matrix is stored in *string1*.

```
s_element{
        short source_vertex;
        short sink_vertex;
        char initial_string;
        char string1;
        char string2;
        }
```

(Note : In the implementation of this data structure in the program, the two strings storing the most recent and currently calculated symbol string are included in a single two dimensional array.)

4.3.2.2. S-MATRIX

This array of *s-element* data structures stores information required, and results produced by the string multiplication algorithm. The array is two-dimensional square, with subscript equal to the number of resources in the system.

```
s-element    s_matrix[number of resources][number of resources];
```

4.3.2.3. CIRCUIT INFO

This structure stores information on a circuit. The circuit string is stored is stored in *circuit_string*. The format of this string is defined in Section 4.3.2.1. The unique identification number for every circuit is stored in *circuit_number*. The order of the circuit is stored in *order*. A linked list of *circuit_info* data structures stores information on all circuits which occur in the Wait Relation Graph. The data structure stores the address of the next element in the linked list.

```
circuit_info
      char circuit_string;
      short circuit_number;
      short order;
      circuit_info*next;
      }
```

4.3.2.4. PRIMARY CIRCUIT LIST

This data structure is a linked list of *circuit_info* structures. It stores details on all primary circuits occurring in the Wait Relation Graph

```
circuit_info *    primary_circuits;
```

4.3.2.5. NUMBER OF CIRCUITS

This global variable stores the number of primary circuits occurring in the Wait Relation Graph.

```
short         number_of_circuits;
```

4.4. ROUTINES

This section develops each of the algorithms into pseudo code. This is the second stage in the development of the program. The order of the routines follows the order of the algorithms

4.4.1. INITIALIZE_S_MATRIX

This routine creates the S-matrix data structure and initializes each element in the S-matrix. (For a declaration of S-matrix data structure, refer to Section 4.3.2.1.) If a directed arc exists from vertex *i* to vertex *j*, the initial string field in the *ij*th element of the S-matrix is initialized to "*i* (*j*)"; otherwise, the initial string field of the *ij*th element is initialized to NULL.

```
initialize_s_matrix(wait_graph_matrix)
start routine
  s-matrix = allocate(N x N units of s-element), where N is the number
             of resources in system
  For every ij^th element in wait_graph_matrix
    s-matrix[i][j].source_vertex = i
    s-matrix[i][j].sink_vertex= j
    s-matrix[i][j].string1 = allocate space
    s-matrix[i][j].string2 = allocate space
    if ( Wait_graph[i][j].number_of_arcs > 0) then
            s-matrix[i][j].initial_string = " i(j)"
    else
            s-matrix[i][j].initial_string = NULL
    end if
  end for
  return address of s-matrix array to calling routine
end routine
```

4.4.2. FORM_S_MATRIX

This routine is called by the determination_of_all_primary_circuits routine. It forms a symbol matrix $\mathbf{S}^n$ of order i. Equations (6) to (10) are used to form $\mathbf{S}^n$ from $\mathbf{S}$ and $\mathbf{S}^{n-1}$. Recall the declarations of s-element and s-matrix. The ith element of the $\mathbf{S}^{n-1}$ matrix is stored in *s-matrix*[*i*][*k*].*string1*. The *kj*th element of the $\mathbf{S}$ matrix is stored in *s-matrix*[*k*][*j*].*initial string*. The *ij*th element of the $\mathbf{S}^n$ matrix is stored in *s-matrix*[*i*][*j*].*string2*.

```
form_s_matrix(s-matrix, n)
start routine
  copy s-matrix[i][j].string to s-matrix[i][j].string1, for every
  element s-matrix[i][j]
  set all diagonal terms in matrix S^(n-1) to null.
  Calculate every element s_ij^n in the first (N - n) rows of
  matrix [S^n]_ij
```

$$s\text{-}matrix[i][j].string2 = \sum_{k=i+1}^{N} s\text{-}matrix[i][k].string1 * s\text{-}matrix[k][j].initial\_string$$

```
  end for
  Calculate the N-n+1th diagonal element (i= N-n+1) in S^n as,
```

$$s\text{-}matrix[i][i].string2 = \sum_{k=i+1}^{N} s\text{-}matrix[i][k].string1 * s\text{-}matrix[k][i].initial\_string$$

```
  return
end routine
```

4.4.3. END_EXPRESSION

This function advances the string pointer until it reaches the first ')' character at the same level in the string.

```
end_expression(string_pointer)
```

```
start routine
  do
    increment string_pointer
    if current symbol is '('
      Call function end_expression(string_pointer)
    end if
    if current symbol is ')'
      return from algorithm
    end if
    if current symbol is '+' or numeric symbol
      do nothing
    end if
    if current symbol is 'end of string'
      print ERROR MESSAGE and exit from program
    end if
  end do
  return
end routine
```

4.4.4. GET_NEXT_TERM

This function advances the string pointer to the first character past the first occurence of the '+' symbol, ')' symbol or the 'end of string' symbol, all of which must occur at the same level. The function is recursive. In the event of a further branch in the string indicated by a '(' symbol, the algorithm is called again.

```
get_next_term(string_pointer)
start routine
  do
    increment string_pointer
    if current symbol is '('
      end expression(string_pointer)
    end if
```

```
    if current symbol is ')'
      quit and return status = END EXPRESSION
    end if
    if current symbol is '+'
      quit and return status = NEXT TERM
    end if
    if current symbol is 'end of string character'
      quit and return status = END EXPRESSION;
    end if
  while(current symbol is not ')', '+' or 'end of string' character)
  return value of status
end routine
```

4.4.5. GET_CIRCUIT

This function is called by the determination_of_all_primary_ circuits routine. The function extracts all primary circuits in the string passed to it, adds these circuits to the primary circuits linked list and returns the updated list to the calling function. The function implements the algorithm described in Section 4.2.2.2.

```
get_circuit(string, circuit_string, primary_circuit)
start routine
  initialize string_pointer
  do
    increment string_pointer
    if current symbol is '('
      consider the portion of string which lies between string_pointer
      and the end.  Call this rest_line
      do
        get circuit(rest_line, circuit_string, primary_circuit)
      while(return value of get next term(rest_line) is NEXT_TERM )
      add circuit string to primary_circuit linked list
    end if
```

```
    if current symbol is '+'
      consider the portion of string which lies between string_pointer
      and the end.  Call this rest_line
      advance to ')' symbol by calling end expression(rest_line)
      function
      call function get_circuit(rest_line, circuit_string,
      primary_circuit)
      add circuit string to primary_circuit linked list
    end if
    if current symbol is 'end of string' character
      add circuit_string to primary_circuit linked list
    end if
    if current symbol is a numeric character
      add symbol to circuit_string
    end if
  while (current symbol is not 'end of string' character)
 add circuit string to primary_circuit linked list
end routine
```

4.4.6. EXTRACT CIRCUITS FROM DIAGONAL ELEMENTS IN S-MATRIX

This routine is called by the determination_of_all_primary_circuits routine. The diagonal elements of the symbol matrix $\mathbf{S}^n$ of order i are analyzed to extract all primary circuits with i arcs. The primary circuits extracted are added to the linked list of primary circuits and returned to the calling routine.

In accordance with (10), only the first N-m+1 diagonal elements of matrix S^m are analyzed. In this function, the extraction of circuits is done by the get_circuit function. $s[k][k]$.*string2* stores the most recently calculated element of a matrix $\mathbf{S}^n$. This is the string passed to the *get_circuit* function. The routine returns the primary circuits linked list to the calling routine.

```
extract_circuits_from_diagonal_elements(s-matrix, primary_circuits, n)
start routine
  For the first N-n+1 diagonal elements
    extract all primary circuits from the diagonal elements of the
    S^n matrix using the get_circuit routine.
  end for
  return address of primary_circuits linked list
end routine
```

4.4.7. DETERMINATION_OF_ALL_PRIMARY_CIRCUITS

This is the highest routine in the second program segment. It is in charge of identification and extraction of all primary circuits in the Wait Relation Graph of the system. Hence, the role of this function is that of an overseer calling the appropriate functions in order. The final result is a linked list of all primary circuits occurring in the Wait Relation Graph.

The routine first allocates memory for and initializes the S-matrix array. It then creates the *primary_circuit*linked list. In a Wait Relation Graph with N nodes, the maximum number of arcs that can occur in any path is N. Hence the largest order symbol matrix to be computed is of order N. The routine computes symbol matrices up to order N. For symbol matrix $\mathbf{S}^n$, it calls the matrix multiplication and circuit extraction routines. These functions detect all primary circuits with i arcs. The primary_circuits linked list is returned to the calling routine.

```
determination_of_all_primary_circuits(wait_graph_matrix)
start routine;
  s_matrix = initialize_s_matrix( wait_graph_matrix);
  create primary_circuit linked list.
  for every symbol matrix of order from 2 to N
     form_s_matrix( s_matrix, n)
```

```
    extract_circuits_from_diagonal_elements(s_matrix, primary_circuit,
    n)
  end for
  return address of primary_circuit linked list to calling routine
end routine
```

CHAPTER 5
PROGRAM DEVELOPMENT - EXTRACTION OF HIGHER ORDER CIRCUITS

The present chapter explains the development of the third segment in the program--namely, the detection of all higher order circuits in the Wait Relation Graph. This is the third and final segment in the program. The results of the previous chapters, i.e., the list *primary_circuits* and the number of primary circuits, are used in this segment. The chapter starts by describing the theory used in the detection of all higher order circuits. Important data structures are described in Section 5.2. The last section contains the pseudo code descriptions of all routines.

5.1 HIGHER ORDER CIRCUIT DETECTION - BACKGROUND

This chapter explains the theory used in the detection of all higher order circuits. It starts by reviewing the definition of order. Next the development of the algorithm is explained.

5.1.1 ORDER

By definition, *order* is defined to be one less than the number of simple circuits in the closed loop, where each of the simple circuits contributing to the order must intersect any other simple circuit in the closed loop in only one vertex.

Consider the Wait Relation Graph G formed by adding a primary circuit C_0 to a higher order circuit C. Let $C' = C \cup C_0$. By lemma 7, order(C')=order(C)+a, where $a = 1$ if C and C_0 intersect in a single vertex and $a = 0$ if C and C_0 intersect in a path.

5.1.2 THEORY OF HIGHER ORDER CIRCUIT DETECTION

From the theorems in Chapter 2, a manufacturing system is deadlock free if

$$\text{slack}(C, s) > \text{order}(C)\ \forall C \in \mathbf{C}_G \forall\ s\ \varepsilon\ S$$

The order of the circuit is obtained by applying the definition of order to its structure. Knowing every element $C \in \mathbf{C}_G$ and its order we can form the deadlock-free slack conditions. Hence, knowing the elements in set $\mathbf{C}_G$ is an important step in obtaining the deadlock free solution to a system.

Definition.

The set $\mathbf{C}_G^i$ is defined as

$\mathbf{C}_G^i = \{C\text{: order}(C)=i, \text{ and } C \in \mathbf{C}_G\}$.

$\mathbf{C}_G^i$ is called the *set of order i.*

Example 5.1. Figure 5.1 depicts the Wait Relation Graph G of a manufacturing system.

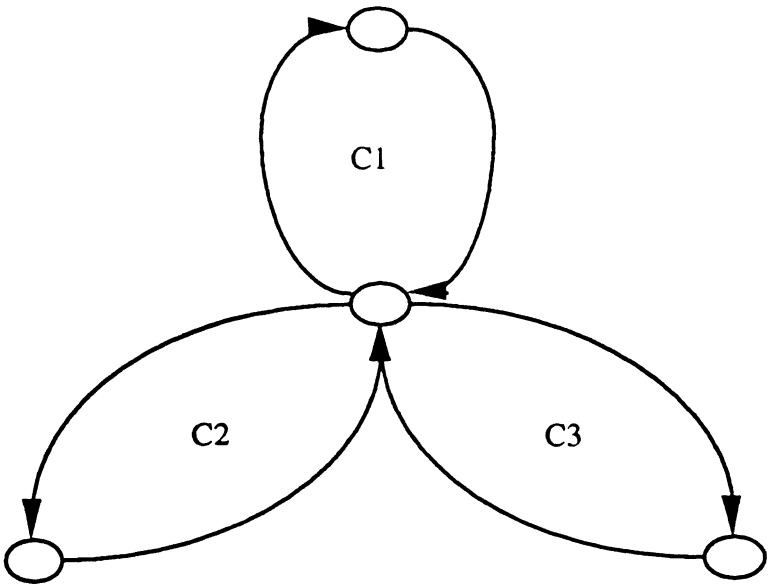

Figure 5.1. The Wait Relation Graph in Example 5.1.

For G we can define the following sets.

The set of order 0, $\mathbf{C}_G^0 = \{C_1, C_2, C_3\}$.

The set of order 1, $\mathbf{C}_G^1 = \{C_1 \cup C_2, C_2 \cup C_3, C_1 \cup C_3\}$.

The set of order 2, $\mathbf{C}_G^2 = \{C_1 \cup C_2 \cup C_3\}$.

$\mathbf{C}_G$ would be obtained as,

$$\begin{aligned} \mathbf{C}_G &= C_G^0 \cup C_G^1 \cup C_G^2, \\ &= \{C_1, C_2, C_3, C_1 \cup C_2, C_2 \cup C_3, C_1 \cup C_3, C_1 \cup C_2 \cup C_3\}. \end{aligned}$$

Given any graph G, the set $\mathbf{C}_G$ can be expressed as,

$$\mathbf{C}_G = \bigcup_{i=0}^{n-1} \cup \mathbf{C}_G^i \text{ , } n = \text{number of resources in system.} \qquad (5\text{-}1)$$

Equation 1 is the basis of an alternate algorithm for deriving $\mathbf{C}_G$. As in (1), $\mathbf{C}_G$ is formed by a union of all sets $\mathbf{C}_G^i$, where $0 \leq i < n$. Each set $\mathbf{C}_G^i$ is formed recursively from the elements in $\mathbf{C}_G^{i-1}$ and $\mathbf{C}_G^0$. The algorithm is based on the material in Section 2.6 and is used in the detection of all higher order circuits. The starting inputs for the algorithm is $\mathbf{C}_G^0$. This is formed from the primary circuits detected by the routines described in Chapter 4. A description of the algorithm follows.

Step 1. Set $i = 0$.

Step 2. Consider an element $C^* \in \mathbf{C}_G^i$.

Step 3. Form the union of C^* with a circuit $C \in \mathbf{C}_G^0$ which does not occur in C^*. Remember that C^* itself can be comprised of multiple primary circuits. The circuit $C \in \mathbf{C}_G^0$ should not be one of the circuits in C^*.

Step 4. If C intersects C^* in a single vertex, order$(C \cup C^*)=i+1$. Add $C \cup C^*$ to $\mathbf{C}_G^{i+1}$. IfC intersects C^* along a path, order$(C \cup C^*) = i$. Add $C \cup C^*$ to $\mathbf{C}_G^i$.

Step 5. Repeat steps 3 and 4 for every circuit C in $\mathbf{C}_G^0$.

Step 6 Repeat step 2, 3 and 4 for every $C^* \in \mathbf{C}_G^i$.

Step 7. Increment i. If $i < n$ repeat steps 2, 3 and 4.

In the proof of Theorem 7(refer to Chapter 2), we know that a Wait Relation Graph can be anaylzed by breaking it up into its primary circuits and simple paths. The graph is then reassembled by adding one element at a time. The above method of building the set $\mathbf{C}_G$ from the union of all sets of orders of orders 0, 1, 2,... n-1 works as well as the method of building the graph by adding simple paths to it, and this is the method adopted in this program.

5.2. DATA STRUCTURES

This section defines the major data structures that are used by the program. They are defined using a C-type notation.

5.2.1. CIRCUIT_INTERSECTION

This structure stores information on the intersection between two circuits. Pertinent information--such as circuit numbers of the two circuits, number of circuit intersections, and any one common vertex (*first_common_vertex*)--are stored.

```
circuit_intersection{
  short     circuit1_number
  short     circuit2_number
  short     number_of_common_vertices
  short     first_common_vertex
  }
```

5.2.2. INTERSECTION

This is an array of circuit_intersection data structures. The array is of size N_C x N_C, where N_C is equal to the number of primary circuits. The *ij*th element of the array stores the information on intersections between primary circuits *i* and *j*.

```
circuit_intersection intersection[N_C][N_C]
```

5.2.3. HIGHER_ORDER_CIRCUITS

This data structure is a linked list of circuit info structures. It stores details on circuits of all orders greater than 0.

```
circuit_info *    higher_order_circuits
```

5.2.4. HIGHEST ORDER CIRCUITS

This data structure is a linked list of circuit_info structures. It stores details on those circuits which are of highest order.

```
circuit_info *    highest_order_circuits
```

5.2.5. CURRENT_CIRCUITS

This is a temporary linked list of circuit_info data structures.

```
circuit_info *    current_circuits
```

5.2.6. NUMBER OF CIRCUITS

This global variable stores the number of primary circuits.

```
short number_of_circuits
```

5.3. ROUTINES - PART 1, FORMATION OF "INTERSECTION" ARRAY

The pseudo code implementation is done in two layers. At the lowest layer is a set of utilities which construct the *intersection* array. This section describes this set. Routines are described in a bottom up approach. The last routine uses all the routines described prior to it.

5.3.1. GET_CIRCUIT_STRING

This routine accepts as a function parameter a circuit number. It then scans the linked list of circuits and returns the circuit string corresponding to this number.(Refer to Section 4.3.2.3 for a declaration of the *circuit_info* data structure.)

```
get_circuit_string(number, primary_circuits)
start routine
  for every circuit in list primary_circuits
      if ( number equals circuit_number field)
            return circuit_string
      end if
  end for
  return
end routine
```

5.3.2. ACT_ON_STRING

This routine performs a variety of operations on a string. These include extraction of the first node in the string, and advancing the string pointer to first node beyond the next white space. The routine returns a string containing the result of these operations. In case the end of string is encountered, the routine returns the NULL string. The notion of a string pointer is similar as in Chapter 4. The string pointer stores the location within the string where the next string will be extracted. The action to be performed is included in a parameter *action_code* passed down by the calling function.

```
act_on_string(ckt_string, action_code, string_pointer)
start routine
  if (action_code = move string pointer to first node after the next
      whitespace in ckt_string)
      increment string_pointer till it is beyond the first ' ' character
      Consider the part of the string which lies between the
            string_pointer location and the end of the ckt_string.
            Update ckt_string to be this string
      if (there are no ' ' characters in ckt_string)
            return NULL string
      else
            return ckt_string
```

```
      end if
  end if
  if (action_code = get next node in ckt_string)
      if (there are no ' ' characters in ckt_string)
            print error message and exit program
      else
            increment string_pointer till it is beyond the first ' '
                  character
            return first node in ckt_string  (this will be the string
            which lies between the whitespace just passed and the next
            whitespace).
      end if
  end if
  return NULL string
end routine
```

5.3.3. CREATE_INTERSECTION_ARRAY

This routine creates the *intersection* array. For every *ij* th element in the array, the routine calculates the number of circuit intersections. It does this by extracting each vertex in circuit *I* and checking for its occurrence in circuit *j*. The number of such occurrences is counted and stored in the circuit intersection data structure corresponding to the *ij*th element of the intersection array. The pseudo code description follows.

```
create_intersection_array( primary_circuits)
start routine
  initialize every element in the intersection array
  for every pair of primary circuits
      Call the pair ( A, B) where A and B are primary circuits
      Determine the number of common vertices between A and B.  Use
            function act_on_string to determine the vertices for each
            circuit
      Store this information in the ijth element of the intersection
            array
```

```
  end for
  return
end routine
```

5.4. ROUTINES -PART 2, DETECTION OF HIGHER-ORDER CIRCUITS

This section contains the pseudo code descriptions for two routines, which together detect all higher-order circuits. The first routine detects all circuits of order 1. The next routine detects all circuits of orders greater than 1. Both routines rely on the intersection array as the source of information on primary circuit intersections. The description of these routines concludes this chapter, as well as the description of the program.

5.4.1. DETECTION_OF_1ST_ORDER_CIRCUITS

This routine detect all 1st order circuits. The intersection array is examined to obtain all pairs of circuits intersecting in a single vertex. A union of every such pair of circuits is formed. From the definition of order, this union represents a circuit of order 1. A list higher-order circuits, consisting of all circuits of order 1, is formed. The pseudo code description follows.

```
detection_of_1st_order_circuits(intersection, primary_circuits)
start routine
  for  every pair of primary circuits
    Call the pair ( A, B) where A and B are primary circuits
    if ( the pair of circuits intersects in a single vertex - this
      information is obtained from the intersection array) then form a
      circuit C = A B
            Add C to the list higher order circuits
    end if
```

```
  end for
  return address of higher_order_circuits
end routine
```

5.4.2. DETECTION_OF_ALL_HIGHER_ORDER_CIRCUITS

This routine detects all circuits of order greater than 1. The routine is based on the algorithm described in Section 5.1. First define a variable *order_of_highest_circuit* and initialize it to 1. Choose an element A from the *highest_order_circuits* list(initially this equals the list of first order circuits). Next, choose an element B from the list of primary circuits. If B intersects A, a new circuit $C = A \cup B$ is formed. By the definition of order, it is known that when a primary circuit intersects another circuit in a single vertex, a circuit of 1 higher order is created. By the same definition, when the circuit intersects another circuit in a path, the order of the resultant circuit does not increase. The order of C is determined by applying these rules. The circuit C is added to a list *current_circuits*.

The entire process outlined in the previous paragraph is repeated once again for every pair of circuits (A, B), where A is from the *highest_order_circuits* list and B is from the list of primary circuits. The list of *current_circuits* is appended to two existing lists. The first is the list *higher_order_circuits*. The next is the list *highest_order_circuits*. The *highest_order_circuits* list is examined; all circuits of order lower than *order_of_highest_circuit*-1 are deleted. The list *current_circuits* is cleared to prepare for a new set of circuits.

At this stage circuits of order = *order_of_highest_circuit* have been detected. The variable *order_of_highest_circuit* is incremented and the procedure repeated. The process is repeated until all circuits of order = *number of resources* - 1 have been detected. At this stage the list *higher_order_circuits* contains all circuits $C \in \mathbf{C}_G$, order(C)>1. The pseudo code description of the routine follows.

```
detection_of_all_higher_order_circuits(higher_order_circuits,
                        primary_circuits)
start routine
  Set the list highest_order_circuits to equal list of first order
      circuits
  Set variable order_of_highest_circuit to 1
  while order_of_highest_circuit is less than the value of
      number_of_resources -1, repeat the steps below
    for every circuit in the list highest_order_circuits
      Call this circuit, A
      for every circuit in the list primary_circuits
            Call this circuit, B
            if( A intersects any circuit in B - determine this
                        information from the intersection array)
                  create a new circuit C = A B
                  if( A intersects B in a single vertex -
                        determine this information from the intersection
                        array)
                        order C = order A + 1
                  else
                        order C = order A
                  end if
                  add C to the list current_circuits
            end if
      end for
    end for
    Add all circuits in list current_circuits to higher_order_circuits
    Add all circuits in list current_circuits to
          highest_order_circuits
    Clear list current_circuits
    For every circuit C in list highest_order_circuits
          if (order of circuit C < order_of_highest_circuit-1)
                delete circuit C from list highest_order_circuits
          end if
    end for
```

```
    increment variable order_of_highest_circuit
  end while loop
  return list higher_order_circuits to calling routine
end routine
```

5.4.3. PRINT_SLACK_CONDITIONS

This routine analyzes the *primary_circuits* and *higher_order_circuits* linked lists, extracting the slack conditions from the information stored within. This extraction is done in two parts:

1. The primary circuits slack conditions are extracted from the information stored in *primary_circuits* linked list. From each linked list element (refer to Section 4.3.2.3), the circuit string(C) and its order are extracted. The slack condition is determined from the relation

$$\text{slack}(C, s) < order.$$

The circuit C is analysed and every node in it extracted. Express these nodes as $N_1, N_2, \cdots, N_m$.

2. The higher-order slack conditions are extracted from the information stored in *higher_order_circuits* linked list. From each linked list element, the circuit string(C) and its order are extracted. The circuit C is itself a union of primary circuits. These are extracted and expressed as

$$C = C_1 \cup C_2 \cdots \cup C_n .$$

The slack condition is

$$\text{slack}(C, s) < order.$$

```
print_slack_conditions(primary_circuits, higher_order_circuits)
start routine
  for every element in primary_circuits
      Extract every node in circuit string
      print circuit and order as described above
  end for
  for every element in higher_order_circuits
      print circuit and order as described above
  end for
end routine
```

CHAPTER 6
ANALYSIS OF CIRCUITS USING THE PROGRAM

In this chapter, the program will be used to analyse a series of examples and results will be presented. The program developed is named `thesis` and resides on SUN workstation *phoenix.ent.ohiou.edu*. The file `system.dat` contains a description of the manufacturing system in the prescribed format. For each example, the Wait Relation Graph of the system, the `system.dat` file and the program output--consisting of a list of circuits with corresponding slack conditions--are presented. The chapter will conclude with a section on how to use the program.

6.1. EXAMPLES

The program is used to analyse a series of examples.

6.1.1 EXAMPLE 1

This example is identical to the one in Section 2.6.1. The process plan is shown in Table 6.1.

Process No.	Process Plan
Process 1	Operation A processes Part 1 in Machine1. Operation B transfers the part via Robot to Machine2. Operation C processes Part 1 in Machine2.
Process 2	Operation D processes Part 2 in Machine2. Operation E transfers the part via Robot to Machine1. Operation F processes Part 2 in Machine1.

Table 6.1. Process Plan of Example 1.

The Wait Relation Graph is shown in Figure 6.1.

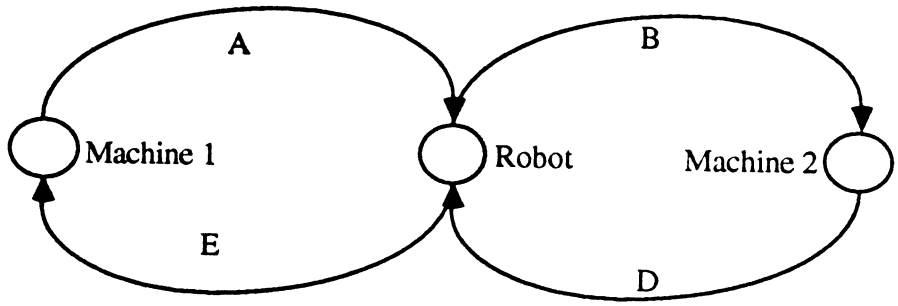

Figure 6.1. Wait Relation Graph of Example 1.

The file `system.dat` is shown in Table 6.2.

RESOURCES
Machine 1 Machine2 Robot
PROCESS1
Machine1 Robot Machine2
PROCESS2
Machine2 Robot Machine1
END

Table 6.2. `system.dat` File for Example 1.

The program output is shown in Table 6.3.

Slack Rule 1 allows atmost 1 operation to be simulaneously active.
The operations are: E, A.
Slack Rule 2 allows atmost 1 operation to be simulaneously active.
The operations are: B, D.
Slack Rule 3 allows atmost 1 operation to be simulaneously active.
The operations are: A, B, D, E.

Table 6.3. Output from Program for Example 1.

6.1.2. EXAMPLE 2

This example is identical to the one in Section 2.6.2.

The process plan is shown in Table 6.4.

Process No.	Process Plan
Process 1	Operation A processes Part1 in Machine1. Operation B processes Part1 in Machine2. Operation C processes Part1 in Machine3. OperationD processes Part1 in Machine4. Operation E processes Part1 in Machine5
Process 2	Operation F processes Part2 in Machine5. Operation G processes Part2 in Machine3. Operation H processes Part2 in Machine4. Operation I processes Part2 in Machine2. Operation J processes Part1 in Machine1.

Table 6.4. Process Plan of Example 2.

The Wait Relation Graph is shown in Figure 6.2.

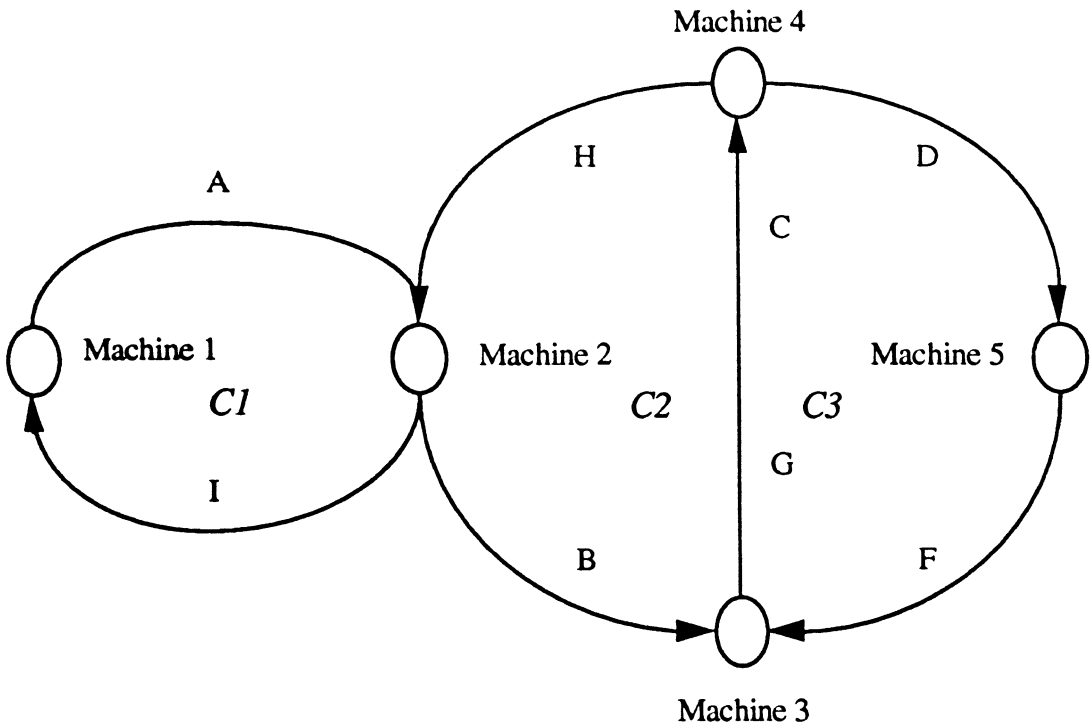

Figure 6.2 Wait Relation Graph of Example 2.

The file `system.dat` is shown in Table 6.5.

RESOURCES
Machine1 Machine2 Machine3 Machine4 Machine5
PROCESS1
Machine1 Machine2 Machine3 Machine4 Machine5
PROCESS2
Machine5 Machine3 Machine4 Machine2 Machine1
END

Table 6.5. `system.dat` File for Example 2.

The program ouptut is shown in Table 6.6.

```
Slack Rule 1 allows atmost 1 operation to be simulaneously active.
The operations are: A, I.
Slack Rule 2 allows atmost 2 operations to be simulaneously active.
The operations are: B, C, G, H.
Slack Rule 3 allows atmost 2 operations to be simulaneously active.
The operations are: C, D, F, G.
Slack Rule 4 allows atmost 3 operations to be simulaneously active.
The operations are: B, C, D, F, G, H.
Slack Rule 5 allows atmost 2 operations to be simulaneously active.
The operations are: A, B, C, G, H, I.
Slack Rule 6 allows atmost 3 operations to be simulaneously active.
The operations are: A, B, C, D, F, G, H, I.
```

Table 6.6. Output from Program for Example 2.

6.1.3. EXAMPLE 3

The system consists of four processes sharing five resources, and is more involved than those of previous examples. The process plan is shown in Table 6.7.

Process No.	Process Plan
Process 1	Resource A process Part1 in Operation O11. Resource E process Part1 in Operation O12. Resource B process Part1 in Operation O13.
Process 2	Resource B process Part2 in Operation O21. Resource E process Part2 in Operation O22. Resource C process Part2 in Operation O23.
Process 3	Resource C process Part3 in Operation O31. Resource E process Part3 in Operation O32. Resource D process Part3 in Operation O33.
Process 4	Resource D process Part4 in Operation O41. Resource E process Part4 in Operation O42. Resource A process Part4 in Operation O43.

Table 6.7. Process Plan of Example 3.

The Wait Relation Graph is shown in Figure 6.3.

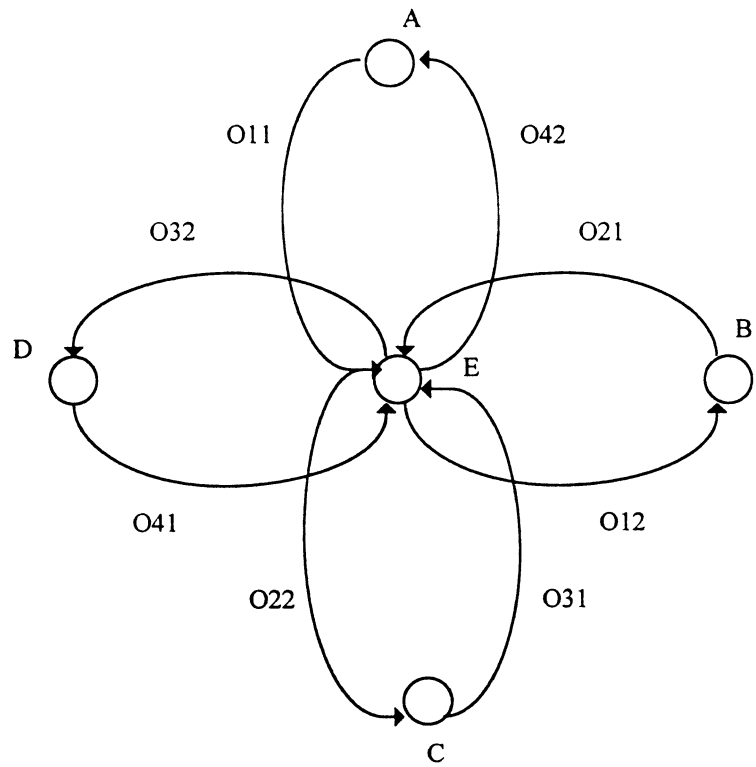

Figure 6.3. Wait Relation Graph of Example 3.

The file `system.dat` is shown in Table 6.8.

```
RESOURCES
A B C D E
PROCESS1
A E B
PROCESS2
B E C
PROCESS3
C E D
PROCESS4
D E A
END
```

Table 6.8. `system.dat` File for Example 3.

The program output is shown in Table 6.9.

```
Slack Rule 1 allows atmost 1 operations to be simulaneously active.
The operations are: O11, O42.
Slack Rule 2 allows atmost 1 operations to be simulaneously active.
The operations are: O12, O21.
Slack Rule 3 allows atmost 1 operations to be simulaneously active.
The operations are: O22, O31.
Slack Rule 4 allows atmost 1 operations to be simulaneously active.
The operations are: O32, O41.
Slack Rule 5 allows atmost 1 operations to be simulaneously active.
The operations are: O11, O12, O21, O42.
Slack Rule 6 allows atmost 1 operations to be simulaneously active.
The operations are: O11, O22, O31, O42.
Slack Rule 7 allows atmost 1 operations to be simulaneously active.
The operations are: O11, O32, O41, O42.
Slack Rule 8 allows atmost 1 operations to be simulaneously active.
The operations are: O12, O21, O22, O31.
Slack Rule 9 allows atmost 1 operations to be simulaneously active.
The operations are: O12, O21, O32, O41.
Slack Rule 10 allows atmost 1 operations to be simulaneously active.
The operations are: O31, O22, O32, O41.
Slack Rule 11 allows atmost 2 operations to be simulaneously active.
The operations are: O11, O12, O21, O22, O31, O42.
```

```
Slack Rule 11 allows atmost 2 operations to be simulaneously active.
The operations are: O11, O12, O21, O22, O31, O42.
Slack Rule 12 allows atmost 2 operations to be simulaneously active.
The operations are: O11, O12, O21, O32, O41, O42.
Slack Rule 13 allows atmost 2 operations to be simulaneously active.
The operations are: O12, O21, O22, O31, O32, O41.
Slack Rule 14 allows atmost 2 operations to be simulaneously active.
The operations are: O22, O31, O32, O41, O42, O11.
Slack Rule 15 allows atmost 3 operations to be simulaneously active.
The operations are: O11, O12, O21, O22, O31, O32, O41, O42.
```

Table 6.9. Output from Program for Example 3.

6.2 USING THE PROGRAM `thesis`

Using the program to analyse manufacturing systems for deadlock is a two step process: preparing the `system.dat` file and running the program. In the first step the `system.dat` file is prepared with a system description. This description is in the format explained in Section 3.1. In the next step, the command `thesis` terminated with a carriage return is typed on the command line. The program executes, printing a list of circuits and corresponding slack conditions.

CHAPTER 7

CONCLUSIONS

In this chapter, goals realized in this thesis and suggestions for future research will be presented.

7.1 CONCLUDING REMARKS AND OBSERVATIONS

In conclusion, formalisms for deadlock avoidance and detection were developed. Theorems 1 to 6 identified deadlock structures, such as circuits and union of circuits. Theorem 7 presented a method to detect deadlock in an arbitrary system based on the results proven in earlier theorems. The concept of slack was introduced. Theorems 2 to 6 used the idea of slack to present a set of conditions which would guarantee a deadlock-free system. Theorem 7 applied the ideas in earlier theorems to an arbitrary system, developing a set of slack conditions for its deadlock-free operation. To test the method developed on a larger variety of manufacturing systems, a computer program was developed. The program was tested on a number of manufacturing systems. Results obtained were consistent with theory.

On comparing the deadlock detection and avoidance method with current research, we found that resource utilizations were higher than PME[3], SME[3], DAA[4] and DAC[5] methods. All deadlock states were correctly identified, when compared to the DDP[1] which ignored some higher-order deadlocks. The deadlock avoidance method developed constraint conditions on groups of operations. These would ensure deadlock-free operation. No buffer resources were used to break deadlock and this was an improvement over Cho's method[2]. However, some non-deadlocked states were

considered deadlocked. The computer program developed for deadlock analysis was entirely offline. No computing resources were diverted for deadlock calculations while the process was running.

7.2 FUTURE RESEARCH

This research provides a reliable base to which many improvements can be made. Each of the suggested improvements are broadly in two categories: those in the first category will enhance performance of the deadlock-free system, while the second will extend the theory to a wider class of systems.

7.2.1 INCORPORATION OF PROCESS PLAN INFORMATION

So far in this research we have ignored all information on specific part flow. In the Wait Relation Graph we were only concerned with the presence or absence of an arc between two nodes. The specific process to which the operation represented belonged was not needed. However, as we will shortly present, this information is useful in eliminating some slack conditions which would otherwise restrict the system and reduce resource utilizations.

Consider once again a two-process, three resource system:

Process 1 Mill Robot Lathe

Process 2 Lathe Mill Robot

The Wait Relation Graph for this system is shown in Figure 7.1.

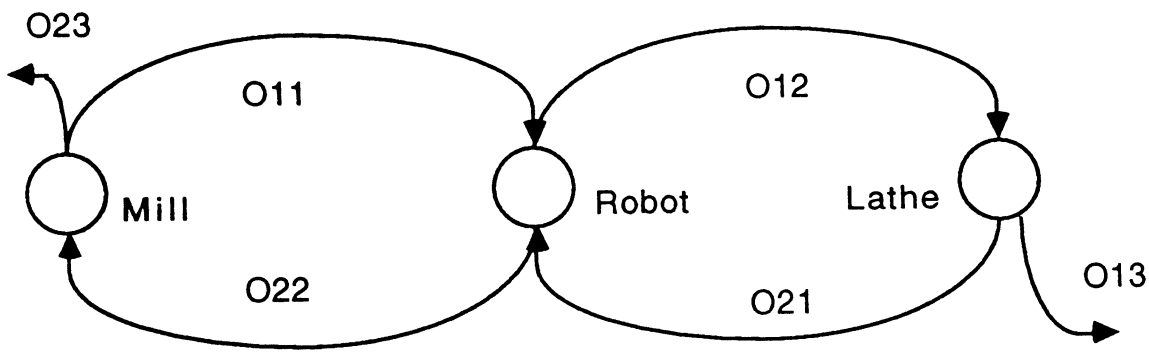

Figure 7.1. WRG for a two-process three-resource FMS.

The following slack conditions are needed for deadlock-free operation:

slack(O11, O22) > 0

slack(O21, O12) > 0

slack(O11, O12, O21, O22) > 1

Now let us closely examine the states of the system eliminated by these slack conditions. This can be best done by listing all possible states row-wise in a table. In Table 7.1 operations are contained in columns with rows depicting states. Each non-zero cell in the table represents an active operation in the corresponding state. The resource used in the operation is entered. A remark is made for each state on whether this state is allowed in the deadlock-free system. For every state disallowed, a remark is then made on whether this is truly a deadlocked state.

Number	O11	O12	O13	O21	O22	O23	Remark	Really Deadlocked?
1	M	0	0	0	0	0	Yes	
2	0	R	0	0	0	0	Yes	
3	0	0	L	0	0	0	Yes	
4	0	0	0	L	0	0	Yes	
5	0	0	0	0	R	0	Yes	
6	0	0	0	0	0	M	Yes	
7	M	0	0	L	0	0	No	Yes
8	M	0	0	0	R	0	No	Yes
9	0	R	0	L	0	0	No	Yes
10	0	R	0	0	0	M	Yes	
11	0	0	L	0	R	0	Yes	
12	0	0	L	0	0	M	Yes	
13	M	R	0	0	0	0	No	No
14	M	R	0	L	0	0	No	Yes
15	M	R	L	0	0	0	No	No
16	0	R	L	0	0	0	Yes	
17	0	R	L	0	0	M	No	No
18	0	0	0	L	R	0	No	No
19	M	0	0	L	R	0	No	Yes
20	0	0	0	L	R	M	No	No
21	0	0	0	0	R	M	Yes	
22	0	0	L	0	R	M	No	No
23	M	0	L	0	R	0	No	Yes
24	0	R	0	L	0	M	No	Yes

Table 7.1. All Possible States in System of Figure 7.1.

From Table 7.1 we can draw the following conclusions:

- All deadlock states were eliminated.
- Some non-deadlocked states were also eliminated.
- The states allowed in the system were all non-deadlocked.
- Of a total of 24 states, 11 were allowed.

- Of a total of 13 disallowed states, 6 were erroneously eliminated

One way to reduce the number of erroneously eliminated states is to include process flow information. For instance, in states 13, 15, 18 and 20 deadlock will not occur, as all processed parts are in only one of the two processes. In states 17 and 22, two parts are either in one of two processes, and a third part is in the last operation in a process. These ideas could be developed and formalized in further research.

7.2.2 ELIMINATION OF ASSUMPTIONS

In Chapter 2 we made the following assumptions on the class of manufacturing systems considered in this research:

- An operation uses just one resource.
- There is one unit of every resource in the system.
- There is no branching of operations in a process plan. Any operation is always preceded/succeeded by a single operation.
- An operation can only process one part at any time.

Elimination of each of these assumptions could be a topic of further research. This will extend the theory to a wider class of manufacturing systems.

BIBLIOGRAPHY

1. Wysk, R.A., Yang, N.S., Joshi, S.M., "Detection of Deadlock in Flexible Manufacturing Cells," IEEE Transactions Robotics and Automation, Vol. 7, No. 6, pp. 853-859, 1991.
2. Cho, H., Kumaran, T.K., Wysk, R.A., "A Graph-Theoretic Deadlock Detection and Resolution for Flexible Manufacturing Systems," IEEE Transactions Robotics and Automation, Vol. 11, No. 3, pp. 413-421, 1995.
3. Zhou, M. C., and DiCesare, F., "Parallel and Sequential Mutual Exclusions for Petri Net Modeling of Manufacturing Systems with Shared Resources," IEEE Transactions Robotics and Automation, Vol. 7, No. 4, pp. 515-527, 1991.
4. Banaszak, Z. and Krogh, B.H., "Deadlock Avoidance in Flexible Manufacturing Systems with Concurrently Competing Process Flows," IEEE Transactions Robotics and Automation, Vol. 6, No. 6, pp. 724-734, 1990.
5. Hsieh, F. and Chang, S. "Dispatching-Driven Deadlock Avoidance Controller Synthesis for Flexible Manufacturing Systems", IEEE Transactions Robotics and Automation, Vol. 10, No. 2, pp. 196-209, 1994.
6. Judd, Robert P. and Faiz, Tariq Nadeem, "Models and Deadlock Avoidance for a Class of Manufacturing Systems", Proceedings of the 1994 Summer Simulation Conference, July 1994, San Diego, pp. 601-606.
7. Judd, Robert P. and Faiz, Tariq Nadeem, "Optimal Deadlock Avoidance of a Class of Manufacturing Systems", Proceedings of the 1994 Automatic Control Conference, June 1994, Baltimore, pp. 707-711.

8. Judd, Robert P. and Faiz, Tariq Nadeem, "Deadlock Detection and Avoidance for a Class of Manufacturing Systems", Proceedings of the 1995 American Control Conference, June 1995, Seattle, pp. 3637-3641.