PATH EXPRESSIONS:

A technique for specifying process synchronization

Roy H. Campbell

Ph.D. Thesis

University of Newcastle upon Tyne, August 1976

ABSTRACT

Path expressions are a new method of describing synchronization which provides a clear and structured approach to the description of shared data and the co-ordination and communication between concurrent processes. This method is flexible in its ability to express synchronization, and may be used in differing forms, some equivalent to P,V operations on counting semaphores.

This method of synchronization is presented, and the motivations and considerations from which it is derived are explained. A method for formally characterizing path expressions is given, together with several automatic means of translating path expressions into implementations using existing synchronization operations.

ACKNOWLEDGEMENTS.

This work was, in part, financed by a grant from the Science Research Council. I gratefully acknowledge the assistance and advice of Professor A.N. Habermann, Dr. H.C. Lauer, Dr. P.E. Lauer, and Professor B. Randell. It was during a series of conversations with Professor Habermann in 1973 that I first developed the idea of a path expression (Campbell and Habermann, 1974). Later, Professor Habermann invited me to Carnegie-Mellon where we finalised the contents of this thesis. Dr. P.E. Lauer contributed to the proof of several of the theorems involving Petri nets, and to the relation between the syntactic classification of path expressions and their Petri net representation (Lauer and Campbell, 1975). Dr. H.C. Lauer and Professor B. Randell were consecutively my supervisors for the thesis, and it was Professor Randell who first suggested the similarity between path expressions and Petri nets.

I would also like to thank Mr. P.M. Melliar-Smith and Dr. D. Lomet for their helpful suggestions and Professor E.S. Page and the members of the Computing Laboratory for their comments and encouragement.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION.

The subject of this thesis is a notation and concept for expressing the synchronization and co-ordination of asynchronous processes. The design and construction of the co-operation of such processes is a difficult task, particularly in large operating systems. Path expressions, first described by Campbell and Habermann (Campbell and Habermann, 1974), provide a high descriptive level of synchronization which aids in the prevention and detection of design errors in complex systems and overcomes some of the hazards, like certain forms of coding errors, that arise in the use of primitive methods involving, for example, P and V (Dijkstra, 1968a) or Wait and Signal (Habermann, 1972).

We shall consider a process as operating, by a sequence of actions, on a known set of objects. Synchronization and co-ordination are required in order to maintain the integrity of such objects shared among different processes. Primitive synchronization methods spread the implementation of such synchronization and co-ordination throughout the programs of various concurrent processes. Monitors (Hoare, 1974), Secretaries (Dijkstra, 1973) and message passing systems (Brinch Hansen, 1972) can be viewed as a step in the direction of associating the specification of synchronization required with the shared object. For example, monitors enforce mutual exclusion on the set of operations or entry procedures which may access the shared data they contain. However, co-ordination between processes using the monitor is accomplished by programming, within the entry procedures, the interleaving of executions of those processes using a sophisticated wait and signal primitive (Hoare, 1974), (Brinch Hansen, 1972).

The path notation allows the specification of the synchronization and co-ordination of actions to be made independently of the descriptions of the processes. To synchronize two actions, each must be provided by a separate procedure invocation. Synchronization and co-ordination are specified directly by describing how the body of one procedure, as a unit, is allowed to execute in relation to others, irrespective of when invoked by processes. The notation for specifying this is called a path expression.

Path expressions may be used to incorporate synchronization rules into the type definitions that are used to introduce each class of object (Campbell and Habermann, 1974). The primary function of a type definition in a program is to describe the implementation of the operations on objects of this type in terms of earlier defined operations on the structural parts of such objects. Together, the type definition and path expressions describe which procedures (operations) may be invoked by a program to access data objects and how these procedures are to be synchronized and co-ordinated to allow the objects to be shared among separate processes. Shared data objects created from such type definitions appear as atomic entities at the places where used, the details of the implementation of the object being irrelevant and underlying structure not accessible.

In this chapter, we shall introduce our proposal using an example notation (Campbell and Habermann, 1974) described informally with the aid of examples. Then, we shall outline our notion of type and discuss how it complements the synchronization method. A brief description of an implementation for the notation will be given and we will show that it may express any synchronization given by P,V operations on counting semaphores. Finally, we shall introduce the topics of the later chapters.

1.1 AN EXAMPLE PATH NOTATION

The example notation, although somewhat arbitrary, is based on regular expressions which provide a familiar framework and which allow path expressions to be represented by finite state machines. The notation has been restricted to permit a description of a simple implementation of path expressions by means of other, existing, synchronization primitives. In Chapter 3, we shall formally define a path notation which is similar to this example notation, and we shall enumerate and justify the decisions taken in its construction.

The idea underlying the implementation of the example path expressions can be envisaged as follows:-

> A path expression names the procedures whose execution by processes are to be synchronized. It includes a specification which describes exactly the way in which the synchronization is to be organized. Each path expression is implemented by a <u>controller</u>. Given that an individual synchronized procedure has been invoked by a process, the controller decides when the procedure execution should be allowed to commence, and therefore the process to continue.

The controller mechanism could operate as follows:

> Each procedure commences with a prologue and finishes with an epilogue. A process executing the prologue of a synchronized procedure enquires of the controller whether it may proceed. The controller, using the synchronization specification, may decide to delay execution or to allow it to continue. Finally, when the process executes the epilogue of the procedure, it notifies the controller which may now be able to release other delayed processes.

The notation we introduce in this chapter is designed to simplify the construction of these controllers; we show that they might, for example, take the form of finite state machines constructed from P,V operations and semaphores (Dijkstra, 1968a). In Chapters 5 and 6, we shall introduce notations which require more complex controllers and discuss the merits of the increased power of these notations and the complexity of their implementation.

Synchronization Schemes

Each path notation will contain fundamental synchronization schemes which may be combined by particular rules for the purposes of expressing complex synchronization. The first two fundamental synchronization schemes we shall identify for the example notation are the sequence of actions and the selection from a set of actions. (By action, we mean the execution of a procedure by a process.) A sequence of actions permits each one to occur in the order specified.

Suppose the executions of three procedures p,q and r are to be sequentially synchronized. Then

$$p \; ; \; q \; ; \; r$$

is an example of a path expressions which would, in our notation, express that procedures p,q and r are to be executed one after the other in the sequence given. The procedures may have been invoked by separate processes, in a different order, and with possible intermediate delays. If an invocation of q occurs first, the invoking process will be delayed until procedure p has been executed.

A selection from a set of actions permits only one of those actions to occur.

Suppose the executions of three procedures p,q and r are selectively synchronized. Then

$$p \; , \; q \; , \; r$$

is an example of a path expression which would specify that a selection of one procedure is to be made from p,q or r. The process attempting to execute the procedure selected is allowed to continue, while the processes attempting to execute those procedures not selected are delayed until a new selection is made from amongst p,q and r. The selection is made from amongst those procedures which have been invoked by processes. The selection is made using an unspecified scheduling algorithm which caters for any possible simultaneity.

These two basic schemes may be combined to form more complex path expressions. Thus the path expression :-

p ; (q , r) ; s

synchronizes the executions of procedures p,q,r and s. Executions of q and r are synchronized selectively. The executions of procedure p, the selected procedure from q or r, and the procedure s are synchronized sequentially. Thus, the path allows two possible series of events:-

> Either the execution of p precedes that of q which precedes that of s, or the execution of p precedes that of r which precedes that of s.

For simplicity in the design of controllers for this notation, we allow a procedure name to occur only once in any path expression. This restriction is lifted in Chapter 5 , where, however, this leads to a more complex implementation.

There are two additional synchronization concepts which we find practical to represent. These are <u>repetition</u> and <u>simultaneous execution</u>.

Repetition permits a path expression, once completed, to be repeated. Many processes are cyclic in behaviour and this is reflected in the path notation. In this chapter, we consider an implied loop applying to the whole path expression. The path expression is, itself, contained between the keywords path and end. Again, for simplicity in the design of controllers for our notation, we make the restriction that repeated path expressions may not be embedded within other path expressions. For a number of problems this proves to be sufficient. However, in Chapter 3, we extend the notation to include repetition which may be nested, but in a restricted manner. In Chapter 5, we shall present a general repetition scheme.

Example: The path expression:-

path P end;

synchronizes the procedure P so that it may be executed by processes repeatedly. If many processes invoke P, one of them at a time will be allowed to execute P while the remainder are delayed until their turn comes. The path expression:-

path A , (B ; (C , D)) end;

is an example of a complex synchronization scheme involving the procedures A, B, C and D. At first, either procedure A or B may be selected to execute. If A is selected then, when execution by the process of A is complete, repetition will occur and a new selection made between procedures A and B. If B is selected then, when execution of B by a process is complete, a selection will be made between the procedures C and D. In this case, when the procedure C or D which is selected has been executed by its invoking process, repetition will occur and a new selection made between procedures A and B.

Simultaneous execution permits several processes to execute a given procedure concurrently. In many synchronization problems it is often required that a body of code can be simultaneously executed by several processes provided that by doing so the processes do not infringe other synchronization restrictions. One example of a notation for specifying simultaneous execution with a path expression was introduced by Campbell and Habermann (Campbell and Habermann, 1974). This consists of a bracket { } placed around the path expression. Nesting of these brackets is not allowed. With this synchronization scheme, as many instances of the enclosed expression as there are requests for it are generated until all instances have been completed.

For example, the path expression:-

{A}

synchronizes the procedure A so that it may be executed by many processes simultaneously. Once one process begins to execute A, other processes may do the same without delay, provided that there are outstanding (uncompleted) executions of A. As soon as the last of these is finished, the simultaneous execution is considered to be completed, and further processes invoking A will be delayed. The path expression:-

<u>path</u> A ; { B ; C } <u>end</u>;

synchronizes the procedures A, B and C using a combination of several basic synchronization schemes. Procedure A may be executed first by a process. On completion of procedure A by a process, the sequence B ; C can be executed by many processes simultaneously. A process invoking procedure C will be delayed until the execution of procedure B is completed by some other process. Procedure B and C may be executed simultaneously by many processes. However, the number of processes executing and which have executed procedure C can never exceed the number of processes which have executed procedure B. If, at some time, all requests by processes for the sequence B ; C have been completed (the number of executed procedures B equals the number of executed procedures C and there are no more invocations of B), then repetition will enable a new invocation of A by a process to execute.

This scheme of simultaneous execution has proved difficult to formalize. Further discussion of this scheme and other more amenable and less restrictive notations are discussed in Chapter 7.

Path Expression Examples

In our opinion, path expressions provide a clear and compact method for describing synchronization problems. For example, the path expression:-

path read , write end;

specifies a series of executions by processes of the procedures read and write in unpredictable order, none of which overlap in time.

The path expression:-

path {read} , write end;

specifies a series of executions by processes of the procedures read and write in unpredictable order. Read executions may overlap other read executions, but write executions may not overlap other read or write executions. Reading, once started, will continue for as long as there are processes invoking read and at least one process executing read.

The above path specifications can be used, for instance, for programming file processing, and we shall now demonstrate how they may be adapted so that a particular access priority can be implemented and localized in one place. In the last example, once reading commences, all processes requesting reading may proceed. It is therefore conceivable that one wants a policy in which, once writing commences, all processes requesting writing will proceed, provided they do so one at a time. This can be implemented by means of two path expressions:-

```
        path {read} , {write} end;
        path write' end;
```

where write is the procedure defined by:-

```
        procedure write: begin write' end;
```

The procedure write' actually performs the writing action. The first path ensures that if the read procedure begins to execute, all reading requests are accepted, and similarly for write. The second path ensures that the actions of writing are mutually exclusive. Thus, executions of write are synchronized with respect to the first path, and executions of its body (write') are synchronized with respect to the second path. The synchronization specification given by each path can be understood separately since, in the notation of this chapter, a particular procedure name can appear in only one path expression. (In other words, there can be a separate controller for each path.) In the paper which first described path expressions (Campbell and Habermann, 1974) further examples of controlling the access to a file using path expressions were given. In particular, it was shown that strategies could be implemented, for example, letting processes requesting writing have priority over processes requesting reading (see Courtois, Heymans and Parnas, 1971). The use of synchronized procedures invoking further synchronized procedures is discussed in greater depth in Chapter 4.

The examples above and in the papers by Campbell and Habermann and Habermann (Habermann, 1975), serve to demonstrate the synchronization facilities that even a simple path notation may provide and illustrate their power. We shall return to this point in the next section where we show that the above notation can be used to program P and V operations on semaphores. In later chapters we shall discuss more formally the equivalences and differences between path notations and between path notations and primitive synchronization operations. While not actually proving the correctness of the path expressions above, we have been able to make assertions about their behaviour. Throughout the thesis we shall relate our path notations to a mathematical representation based upon Petri nets (Petri, 1962). This allows, wherever permitted by Petri net theory, a formal evaluation of the correctness of a set of path expressions with respect to the synchronization and co-ordination which they are intended to express. In addition, we develop a model to investigate the behaviour of programs in which the paths are used.

1.2 STRUCTURING SYNCHRONIZATION WITH TYPES.

Types, under one name or other, have been used in programming languages for many purposes. Notable instances of their use are in "type checking" as in Pascal (Wirth, 1971), for providing mutual exclusion as in Concurrent Pascal (Brinch Hansen, 1974), for describing building blocks as in extensible languages (for example ECL (Wegbreit, 1972)) and for implementing new data objects and operations as in the classes of Simula 67 (Dahl, 1970) or the modes of Algol 68 (Algol 68, 1968).

We believe that type definitions should be used to accomplish on behalf of the construction of data objects the analogue of what procedure declarations do for operations. Thus, at the place where it is used, the details of the implementation of an object should be irrelevant and underlying structure should not be accessible at that moment. The reasons why this should be so are obviously the same as those that underlie a procedure mechanism: separation of specification and implementation, and concentration of implementation issues so as to facilitate verification, debugging and modifications.

As a natural consequence of this point of view, a type definition is used to create new data objects which appear as atomic entities at the places where used. Drawing the parallel between type definitions and procedure declarations, accessing a structural part of a typed object is similar to jumping into a procedure body.

The primary function of a type definition in a program is to describe the implementation of the operations on objects of this type in terms of earlier defined operations on the structural parts of such objects. Thus, another part of a type definition ought to be a description of the detailed structure that objects of this type will have.

These two functions of a type definition are reflected in a notation which we have devised to help us in our ideas. The syntax is purely suggestive and is not intended as a proposal for a new programming language or any part of one.

For example we might have:-

```
type buffer;
message frame;
     procedure read (returns message m): m:=frame;
     procedure write (accepts message m): frame:=m;
operations read,write
endtype
```

This example is a definition of a type called buffer, whose structure consists of a variable frame of type message, and whose operations are the procedures read and write. (The type message is assumed to have been previously defined.) Instances of buffers can be declared or created in the scope of the type definition, and each one will contain its own instance of frame. The program using a buffer cannot, however, access frame directly but must use <u>read</u> and <u>write</u>. The procedure can be applied to them by means of the Simula 67 dot notation.

For example:-

```
buffer A;
message T;
T:= A.read;
A.write (T);
```

The type definition has two important properties:-

1) Protection of its structure by the scope rules.

2) Only a fixed, identifiable set of procedures is defined, giving carefully controlled access to the data of the objects of that type.

Objects created from type definitions can be common to the scope of two or more processes. However, the type buffer defined above is not satisfactory when various concurrent processes may simultaneously read and write a shared buffer, and some form of synchronization is required.

In general, the sharing of objects of a type will be unsatisfactory if the data contained in the object can be corrupted by several processes executing procedures simultaneously or in invalid sequences. Monitors (Hoare, 1974) are an attempt to prevent corruption by allowing only one process at a time to access and alter the contents of an object. Prevention of the occurrence of invalid sequences of actions on the object or its contents must be programmed, however, through the use of a signal and wait primitive. We will combine our path expressions with our notion of type to introduce some orderly structure in the sharing of objects.

The restrictions we must place on the operations read and write of our buffer example in order to preserve the integrity of the contained data are:-

1) Every read must be followed by a write.

2) Every write must be followed by a read.

3) A read and write must not occur simultaneously.

Provided the buffer obeys these rules we can assert that it will not lose or destroy any information.

The following type ensures these three properties:-

```
type buffer;
message frame;
path write ; read end;
        procedure read (returns message m):  m:=frame;
        procedure write (accepts message m):  frame:=m;
operations read, write
endtype
```

The path is applied to the operations to produce the correct synchronization. A different instance of the synchronization path is associated with each instance of a buffer. Thus any declaration of a buffer will result in an atomic object which can be written or read alternately. In Chapter 4 we shall make a more formal definition of the relationship between a type and the paths it contains as part of its specification.

This type definition of a buffer may be used to build more complex data structures, for example objects of type "ring buffer". Suppose that a number of similar readers and writers wish to exchange information but are constrained by the amount of space available for buffers. One such device is demonstrated below and is designed to permit as much concurrency as possible. A ring of the above described buffers is declared. A send or receive request allocates a buffer to be read or written on a round-robin basis. The integrity of the individual buffers is assured by their type definition. Allocation is achieved by advancing pointers around the ring of buffers. Many requests to send or receive may occur simultaneously. However, each pointer may only be advanced by one process at a time if the integrity of the allocation mechanism is to be preserved. A type pointer is introduced which includes the necessary synchronization. Thus we have:-

```
type ring-buffer;
array 0 to N-1 buffer R;
        type pointer;
        integer P=0;
        path next end;
                procedure next (returns integer I):
                     begin P:=(P + 1) mod N; I:=P; end;
        operations next
        endtype;
pointer write_slot, read_slot;
        procedure send (accepts message m):
                begin integer j; j:=write_slot.next;R(j).write(m); end;
        procedure receive (returns message m):
                begin integer j; j:=read_slot.next;m:=R(j).read; end;
operations send, receive
endtype;
```

The implementations of the buffers and the pointers are separated from the ring-buffer mechanism. Similarly, the readers and writers can be programmed independently of the implementation of the buffering system.

1.3 A COMPARISON BETWEEN A PATH NOTATION AND P,V OPERATIONS ON COUNTING SEMAPHORES

The examples we have described above encourage us in our belief that the method is worth studying and is a potential contribution to better-structured and safer synchronization methods.

We shall now show that the synchronization mechanism described above may express any synchronization constructed with the more primitive synchronization operations such as P and V operations on counting semaphores or Signal and Wait (Dijkstra, 1968a), (Habermann, 1972). This provides an illustration of the ability of a path notation to express synchronization. For example the path:-

```
                path {V ; P} end
```

provides the synchronization necessary to implement P and V operations. The number of executed Ps can never be greater than the number of completely executed Vs. This path may be embedded within a type description for a semaphore. Thus:-

```
    type semaphore;
    path {V ; P} end;
            procedure V: null;
            procedure P: null;
    operations P,V
    endtype;
```

Variables of type semaphore may be declared and each instance will have its own synchronization path. The 'value' of the semaphore can only be changed by executing either a P or a V. In the example above, semaphores are always initialized to zero. An extension to the above notation would be to include initialization in types and perhaps in paths. Thus a program restricted to using our notation has lost none of the potential for expressing synchronization that the use of P,V operations on counting semaphores would have given, but has gained the structuring facilities that the use of types and paths provide.

1.4 IMPLEMENTATION.

One important aspect of our example notation is that it has a practical implementation. Controllers for our notation can be implemented using existing synchronization methods and these may be generated automatically from the path expressions, for example by a compiler. We will show one particular implementation in which path expressions are transformed into appropriate P and V operations on counting semaphores for use in prologues and epilogues of the procedures the path expressions name. A complete description of the algorithm to transform these path expressions is given in (Campbell and Habermann, 1974). (incidently, this completes the equivalence between the two synchronization methods.) The following example illustrates the translation of a path expression into P and V operations.

The path expression:-

```
        path A ; ( B , (C ; D) ) ; E end;
```

would result in the following set of prologues and epilogues:-

```
semaphore s1,s2,s3,s4;
s1:=1;  s2:=s3:=s4:=0;
procedures
A : begin  P(s1);    (body of A);    V(s2);    end;
B : begin  P(s2);    (body of B);    V(s3);    end;
C : begin  P(s2);    (body of C);    V(s4);    end;
D : begin  P(s4);    (body of D);    V(s3);    end;
E : begin  P(s3);    (body of E);    V(s1);    end;
```

Initially, any processes invoking a procedure B,C,D or E will be suspended by the P operations in the prologue of that procedure. A process invoking an A will, however, be able to decrement semaphore s1, reducing it to zero, and thus preventing further executions of an A. On completion of the A, the process increments s2 in the epilogue of the procedure. This permits one of procedures B and C to be executed by a process. The selection is implemented by the processes invoking B and C competing over performing a P operation on s2. If procedure C succeeds, then the epilogue of C, when executed, will increment s4. This permits a D to be executed. A process executing the epilogue of D will increment s3. (Similarly, if the procedure B had been executed instead of a C, then the process executing it would increment s3 in the epilogue of that procedure.) If s3 is incremented, this permits a process to execute procedure E, and will result in s1 being incremented in the epilogue of that procedure. This gives the synchronization of the repetition which allows another A to be executed.

The example described above serves to show that the path expression provides a structured synchronization technique which emphasises what is needed, not how it is to be achieved. The P,V implementation of the path expression does not directly express the synchronization it is used to create. See also (Brinch Hansen, 1972). Therefore, our mechanism can lead to automatically generated, well structured uses of synchronization primitives and the programmer is relieved of the problem of implementing his desired synchronization.

1.5 INTRODUCTION TO THE MATERIAL IN THE THESIS.

We have introduced a new method of synchronization which provides a clear and structured approach to the description of shared data and the co-ordination and communication between concurrent processes. As an example of this method, we have outlined a notation which we demonstrated had these properties and was equivalent in its ability to express any given synchronization to P and V operations on counting semaphores.

In the remainder of the thesis we shall expand and develop these ideas and show that there are many possible Path notations. In Chapter 2 we shall introduce notation which will be used in the thesis. Chapter 3 describes the E-path notation which is a development of the notation above. We shall introduce two measurements of the power of synchronization of a given synchronization notation which we shall use in later chapters to compare different notations. Chapter 4 establishes a mathematical meaning for the combination of path expressions with Petri net models of processes. We shall show that the implementation of the E-path expressions is correct with respect to this meaning. The formalism of Chapter 4 enables us to give a simple criterion for a limited class of Path expressions and processes to be deadlock free. In Chapter 5 we shall generalize the notation of Chapter 3 and describe the meaning and an implementation for this more powerful descriptive method. In Chapter 6 we shall introduce the possibility of describing the synchronization of individual actions concurrently, by separate synchronization specifications. Finally, in Chapter 7 we shall discuss various Path notations which differ from those presented in the body of the thesis, and describe the problems and advantages of these notations and how they reflect on the theme of the thesis.

CHAPTER 2

BACKGROUND.

The following notation and definitions will be used in the remainder of the thesis. This chapter contains a description of Petri nets, some definitions and theorems from Petri net theory, and definitions, theorems, and algorithms from the theory of finite state machines.

Petri nets.

Petri nets may be thought of as bichromatic directed graphs in which no two adjacent vertices have the same colour. Colour of a vertex is indicated in our diagrams by its shape, either round or square (we shall abbreviate a square by a line). Round vertices and square vertices are called places and transitions, respectively, and denote conditions and events. A net, being a set of vertices interconnected with directed arcs, can be used to describe arbitrary systems of events and their occurrences. This interpretation is obtained by specifying that an event may occur if all its enabling conditions hold in the system. This is indicated in the net by drawing a directed arc from the places denoting the enabling conditions of the event to the transition denoting the event. The occurrence of an event may subsequently cause other enabling conditions to hold; this is represented by drawing arcs from the transition representing the event to the places denoting the conditions. The net can be used to simulate system behaviour by putting tokens in places of the net to indicate the holding of conditions. A transition is enabled to fire when all its places are marked with such tokens. The firing of a transition subtracts a single token from each enabling place and adds a single token to each place which is on a directed arc from the transition (the output place). Thus, the net specifies the set of all possible sequences of concurrent events of the system.

If two transitions share an enabling place with one token on it, they are said to be in conflict over that place; both are enabled but only one will fire disabling the other. This implies that a marked net represents the set of all sequences of occurrences of events (firings of transitions) determined by the holding of the conditions corresponding to the marked places.

The following definitions will be assumed:-

2.1) A marked net is live if each of its component transitions is live (Hack, 1972).

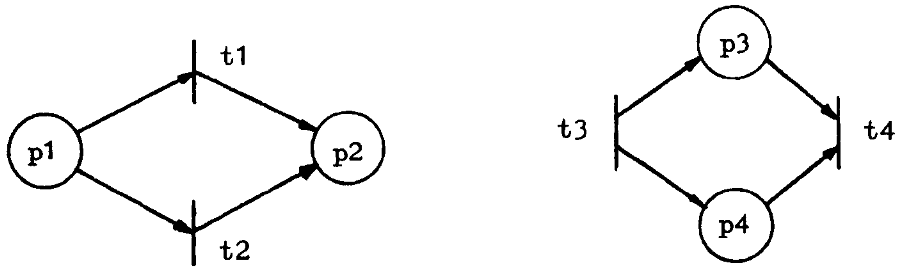
2.2) A transition is live at a given marking if there exists a sequence of firings which fires it for every marking reachable from a given marking. (That is, an event is live, given the holding of some set of conditions, if those conditions may eventually allow it to occur.)

2.3) A marking is reachable from another marking if there exists a sequence of firings which transforms it from the latter into the former.

2.4) A marked net is n-safe (safe) if all its component places are n-safe (safe), (Petri, 1962).

2.5) A place is n-safe (safe) at a given marking if every marking reachable from the given marking has at most n (one) tokens on that place. (I.e., a condition is safe if, given the holding of some set of conditions, two or more instances of the holding of that condition may never occur simultaneously.)

2.6) In the diagram below:-



p1 and p2 are shared input and output places, respectively, of t1 and t2.

t3 and t4 are shared initial and terminal transitions, respectively, of p3 and p4.

2.7) A State Machine Petri net has shared input and/or output places, but no shared initial or terminal transitions.

2.8) A State Machine Petri net is strongly connected if and only if every vertex in the net is 'connected' to every other vertex by some sequence of directed arcs and vertices (Holt and Commoner, 1970).

2.9) A Marked Graph Petri net has shared initial and/or terminal transitions, but no shared input or output places.

2.10) A Simple Petri net is one in which every transition has at most one shared input place (Hack, 1972).

The following properties of the State Machine and Marked Graph Petri nets have been proved by Holt and Commoner (Holt and Commoner, 1970) and will be used later in the thesis:-

2.11) A strongly connected State Machine Petri net is live and safe for any marking which places only one token on the net.

2.12) A transition in a Marked Graph is live if and only if it is not contained in an empty cycle. (A set of vertices and directed arcs is said to be empty if no vertex contains a token, and is a cycle if, by following the directed arcs, all the vertices are visited once and the first vertex is reached again.)

Matrix representations of Petri nets.

To simplify the description of nets and to represent them in a suitable form for computation they may be presented as matrices (Lautenbach and Schmid, 1974). For a given net we have two matrices, a forward incidence matrix (F) and a backward incidence matrix (B). (Hack, 1975). Each column of the matrix corresponds to a transition, each row to a place. When a place has a directed arc to a transition, there will be a one written in the forward incidence matrix at the intersection of the row corresponding to that place and the column corresponding to that transition. Similarly, if a transition has a directed arc from it to a place, the backward incidence matrix will contain a one at the intersection between row and column.

A marking will be represented by a vector (M), each element corresponding to a place containing a number representing the number of markers on that place.

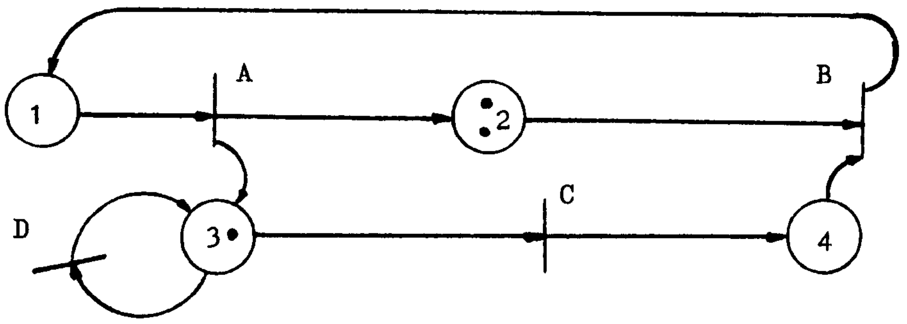
Example:-

	F	transitions			
		A	B	C	D
places	1	1			
	2		1		
	3			1	1
	4		1		

	B	transitions			
		A	B	C	D
places	1		1		
	2	1			
	3	1			1
	4			1	

M
places
0
2
1
0

corresponds to the net:-



The matrix representation is used in later chapters to simplify specifying Petri nets and as a computational tool in the controllers of path expressions. Finally, these matrices have been used in algorithms to investigate Petri net properties and behaviour (Lautenbach and Schmid, 1974).

Finite State Machine Theory.

In Chapter 1 and later chapters we introduce path notations based upon regular expression descriptions of the synchronization of the execution of procedures by processes. We shall examine these regular expression descriptions of synchronization by examining the state machines to which they correspond. The following definitions will be used:-

2.13) A <u>finite state automaton</u> over the alphabet E is a system

$$U = (S,M,So,F)$$

where S is a finite non-empty set, M is a function defined on the cartesian product S $\times$ E of all pairs of states and symbols with values in S, So is an element of S (the initial state of U) and F is a subset of S (the designated final states of U). (M is the table of transitions, and S is a set of the internal states of U.) (See Hopcroft and Ullman, 1969.)

2.14) A <u>non-deterministic finite state automaton</u> over the alphabet E is a system

$$V = (S, N, So, F)$$

where S is a finite non-empty set of states, N is a function defined on the cartesian product S $\times$ E into subsets of S, So is an element of S (the initial state of V), and F is a subset of S (the designated final states of V).

2.15) A transition system over the alphabet E is defined to be a quintuple

$$T = (S, M, A, F, P)$$

where S is the non-empty set of internal states, A is a subset of S and is the set of initial states, F is a subset of S and is the final set of states, M is a transition function which maps S $\times$ E into 2^S, the set of all subsets of S, and P is a binary relation on S which holds between two states u and v in case there is a spontaneous transition from u to v. P is called the spontaneous transition relation. (Harrison, 1965).

2.16) The language of regular expressions, L, is defined inductively as follows:-

(1) Λ is a regular expression.

(2) $\emptyset$ is a regular expression.

(3) σ_i is a regular expression for i = 0,1,...,k-1.

(4) If α and β are regular expressions, then α , β is a regular expression.

(5) If α and β are regular expressions, then α ; β is a regular expression.

(6) If α is a regular expression, then α^* is a regular expression.

(7) There are no regular expressions other than those given by steps (1) to (6).

2.17) The behaviour of a regular expression or the set denoted by a regular expression is defined by the following procedure. (The behaviour of α is denoted by $|\alpha|$.)

(1) $|\emptyset| = \emptyset$, where $\emptyset$ is the empty set.

(2) $|\Lambda| = \{\Lambda\}$.

(3) $|\sigma_i| = \{\sigma_i\}$, for $i = 0,1,\ldots,k-1$.

(4) $|\alpha,\beta| = |\alpha| \cup |\beta|$.

(5) $|\alpha;\beta| = |\alpha|\ |\beta| = \{ xy \mid x \in |\alpha|, y \in |\beta| \}$.

(6) $|\alpha^*| = \{\Lambda\} \cup |\alpha| \cup |\alpha||\alpha| \cup |\alpha||\alpha||\alpha| \cup \ldots$

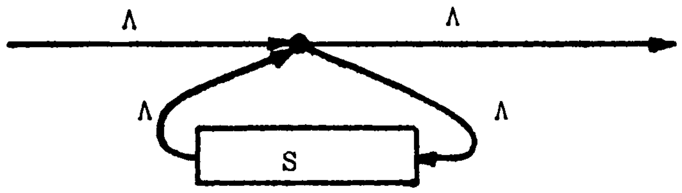
2.18) There is an algorithm which applies to any given regular expression α and yields a finite state machine whose behaviour is $|\alpha|$. (Kleene, 1956).

The following results of finite state machine theory will be used later in the thesis to state and prove properties of path expressions:-

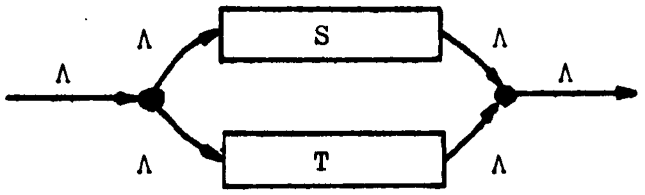
2.19) The following algorithm applies to any regular expression α to yield an equivalent finite state machine. The algorithm first constructs a finite transition system whose behaviour is $|\alpha|$. From this one obtains the appropriate finite state machine by the subset construction (Harrison, 1965).

1) If $\alpha = \emptyset, \Lambda, \sigma_0, \sigma_1, \ldots, \sigma_{k-1}$, then the problem is trivial.

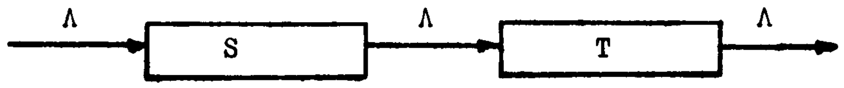
2) If there is a transition system S having behaviour $|\alpha|$, then the transition graph below has behaviour $|\alpha^*|$. (Note that, in the following figures, the single transition into (out of) a transition system symbolizes one line to (from) each initial (final) state.)



3) If transition systems S and T have behaviours $|\alpha|$ and $|\beta|$, respectively, then the transition system having behaviour $|\alpha,\beta|$ is shown below:-



4) If S and T are transition systems having behaviours $|\alpha|$ $|\beta|$, respectively, then the transition system having behaviour $|\alpha;\beta|$ is shown below:-



2.20) The minimum state machine with behaviour $|\alpha|$ is unique up to isomorphism (Hopcroft and Ullman, 1969).

2.21) Strongly connected state machines are state machines such that there is a sequence of transitions from any state to any other state.

We shall now use the above definitions and theorems to introduce and describe the Elementary Path Notation.

CHAPTER 3

ELEMENTARY PATH NOTATION.

In the preliminary discussion in Chapter 1 we introduced the concepts of action, path notation, controllers, and a notion of type. In this chapter, we shall enlarge upon these concepts and give a mathematical representation of expressions belonging to a particular path notation. Various properties of the path notation are discussed, for example, that this particular notation is inherently free from a class of deadlocks. Finally, a detailed discussion of an implementation of this notation is given, together with some comments about the notation itself.

The Elementary Path (henceforth referred to as E-path) notation is a modified form of the notation discussed in Chapter 1. It omits the simultaneous execution scheme which is discussed later in Chapter 7, where it is considered with respect to its inclusion in this and other notations. However, it includes a limited repetition scheme of synchronization.

3.1 MOTIVATION FOR AN "ELEMENTARY PATH" NOTATION.

The E-path notation permits the declaration of a set of specific synchronizations between various actions. This declaration is a static description of the synchronization. This is analogous to keeping the array dimensions constant for the life of an array, to the creation of variables in an Algol block, and to the attempts to eliminate the goto from programming languages. The notation has been designed to be simple, yet sufficiently powerful to be a programming tool, and be easily implementable on most present day computers.

To ensure practicality, E-path expressions have been designed to be implemented by the mechanism described briefly in Chapter 1 using a P operation on some binary semaphore (Shaw,1974) in the prologue of a synchronized procedure, and a V operation on a possibly different semaphore in the epilogue of that procedure. This requirement imposes a restriction on the form that E-paths may take, but nevertheless provides a powerful synchronizing method.

In later chapters, we shall extend this notation and investigate the relations between increasing the synchronization power of the notation (compare Lipton, 1973) and increasing the descriptive ability of the notation. (This may be compared to a low level language and a high level language; both have equivalent computational power, but the latter has more descriptive ability than the former.) In particular, the relations we shall investigate are the manner in which they may be increased, and the effects on comprehensibility and the complexity of the controllers. To achieve these aims we require E-paths to be simple and not to contain any features which may restrict this possibility of expansion.

The E-paths must be consistent with our thesis: that we want to describe synchronization between actions, and that path expressions may be attached to data objects to describe the synchronization which allows those objects to be shared by several concurrent processes. As we shall show, both these requirements are satisfied by the notation, and in addition it helps to prove the validity of our approach.

3.2 CHOICE OF REGULAR EXPRESSIONS AS A VEHICLE.

The form we have chosen to adopt for the E-path notation is based upon regular expressions. The following reasons motivate this decision:-

1) Regular expressions provide a structured framework in which we can write our synchronization descriptions.
2) They are well known, and when considered in the equivalent form

of state machines recognizing strings of symbols, bear a striking resemblance to the synchronization method we require. In addition, the state machine suggests several implementation mechanisms in which it is used as the central part of a controller for synchronizing various actions.

3) They have been used theoretically by others to describe Operating System behaviour, for example, as in Message Transfer Expressions (Riddle, 1972). Petri used a similar notation for his communication forms which described the behaviour of subnets of a Petri net (Petri, 1962). Hence, they are suitably flexible and descriptive. Finally, particular properties of state machines, such as being strongly connected and having minimal canonical forms, can be used in the study and application of path expressions.

The decision to implement the E-path expressions using P and V operations in the prologues and epilogues of procedures implies that the E-path notation can correspond only to a subset of the class of all regular expressions. This subset will be restricted to expressions which can be represented as state machines in which no two transitions share the same name. This enables the states of such state machines to be represented by semaphores. The syntax of E-path expressions is designed to include only regular expressions whose associated state machines meet the requirement of this mechanism. The syntax, although not permitting all the expressions representing regular sets accepted by such state machines, allows the syntactic checking of expressions to be accomplished and readily automated. A detailed explanation of the relationship between the implementation, E-path notation and state machines will follow.

3.3 ELEMENTARY PATH DEFINITION.

The syntax for the E-path notation is defined by an extended BNF notation in which the part in the square brackets of a production is optional. To simplify the description we have assumed an arbitrary precedence relationship between the separators, i.e. the symbols ( ) ; , * representing the various synchronization schemes of the notation. Terminal and non-terminal symbols are distinguished by context, i.e. terminals do not occur on the left hand side of any production. The Kleene star will be used to denote repetition of a subexpression zero or more times. The occurrence of a Kleene star in an expression is restricted by the productions to ensure that the expression has a state machine with the required property permitting implementation.

The construction of an E-path expression is specified by the following five productions and two restrictions on the use of names within the path expression.

```
P1:   E-path        ::=       path ( sequence )* end
P2:   sequence      ::=       unit [ ; sequence ]
P3:   unit          ::=       element * ; element
                          |   selection
P4:   selection     ::=       element [ , selection ]
P5:   element       ::=       procedurename
                          |   ( selection [ ; sequence ] )
```

<u>Restriction R1</u>: No procedurename may occur more than once in a given path expression.

<u>Restriction R2</u>: No procedurename may occur in more than one path expression.

(Restriction R1 will be lifted in Chapter 5, and restriction R2 will be lifted in Chapter 6.)

The productions form an unambiguous grammar and have been used in a recognizer for E-path expressions. Production P1 introduces the path end pair which denotes the path expression, and only permits expressions which are enclosed by repetition. This enforced repetition of the synchronization expression is included for an important theoretical implication which will be described later.

Examples of valid E-paths.

```
path ( open ; ( read ; write )* ; close )* end
path ( openfile ; ( openrecord ; write* ; closerecord )* ;
                                          closefile )* end
path ( openfile ; (
 ( openrecord_for_write ; write* ; closerecord_for_write ) ,
 ( openrecord_for_read ; read* ; closerecord_for_read )
                              )* ; closefile )* end
```

Examples of invalid E-path expressions.

```
path ( A* )* end
path (( A* ; B )* ; C )* end
path ( A* ; ( B* ; C ))* end
path ( A , ( B* ; C ) ; D )* end
```

3.4 E-PATH EXPRESSIONS AS FINITE STATE ACCEPTOR MECHANISMS.

In order to discuss the theoretical properties and describe an implementation for E-path expressions we shall use a finite state machine description of the synchronization they express. Since every

E-path expression is a regular expression, and all regular expressions have equivalent finite state machines (2.18), we may construct an equivalent finite state machine for any E-path expression. We will use a modified version of the construction given by Harrison (see 2.19). The construction involves the generation of a finite transition system (2.15) for a given regular expression. We directly reduce this transition system to a finite state machine by simply eliminating the spontaneous transitions.

<u>Theorem 3.1</u>: Let α be a regular expression such that <u>path</u> α <u>end</u> is accepted by the productions of the E-path notation. The equivalent non-deterministic state machine may be constructed from the transition system for α, generated by 2.19, using the following:-

Let C1 and C2 be sets of states of the transition system:-

$$T = (S,M,A,F,P) \text{ such that}$$

$$C1 = \{ s \mid \bigcup_{z \in E} M(s,z) = \emptyset \text{ and } s \in S \}$$

$$C2 = \{ s \mid (\bigcup_{\substack{z \in E \\ t \in S}} M(t,z)) \cap \{s\} = \emptyset \text{ and } s \in S\}.$$

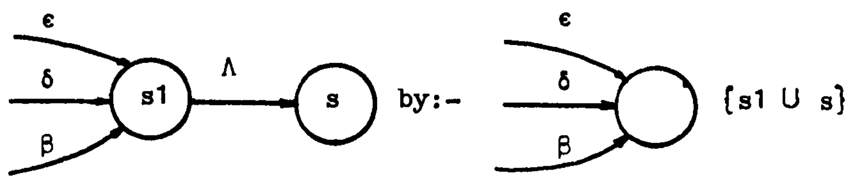
Given s1 is a state of T and if there exists a unique $s \in S$ such that either:-

E1: $P(s1) = \{s\}$ and $s1 \in C1$ (i.e. the only transitions from s1 are spontaneous and go to s), or

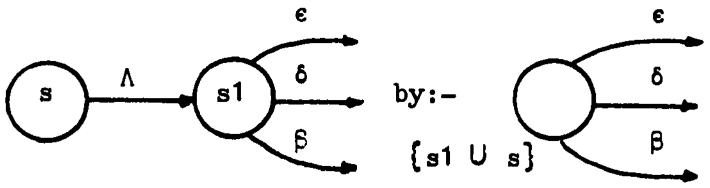
E2: $P(s) = \{s1\}$ and $s1 \in C2$ (i.e. the only transitions to s1 are spontaneous and come from s)

then create a new transition system $T' = (S',M',A',F',P')$ which is identical to the original system except that s1 and s are replaced by {s ∪ s1}. Any transitions to or from s1 or s will become transitions to or from {s ∪ s1} and if s or s1 belong to A (the initial states) or F (the final states), then {s ∪ s1} will belong to A' or F'.

(i.e. replace:-



and

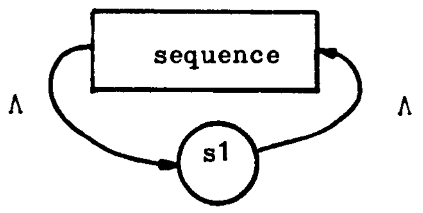


<u>Proof</u>: Trivially, the replacement rules above will not alter the behaviour of the transition system for α.

It remains to be shown that all spontaneous transitions will be removed by these rules. <u>path</u> α <u>end</u> must be produced by the E-path set of productions. We shall apply the transition system construction rules of 2.19 to each production and show, by induction, that the transition system for α has only spontaneous transitions of the form E1 or E2.

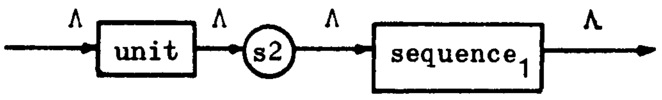
The productions correspond to the following transition systems (T1-T5c).

T1: E-path ::= <u>path</u> (sequence)* <u>end</u>



where s1 ∈ C1 and C2.

T2a: sequence ::= unit ; sequence

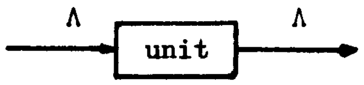


where s2 ∈ C1 and C2.

The initial states are those of 'unit',

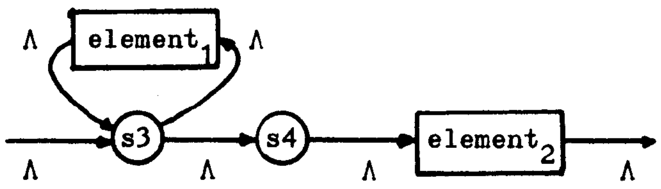
the final states are those of 'sequence$_1$'.

T2b: sequence ::= unit



The initial and final states are those of 'unit'.

T3a: unit ::= element* ; element

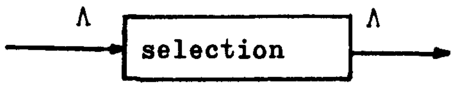


where s4 ∈ C1 and C2.

The initial state is s3,

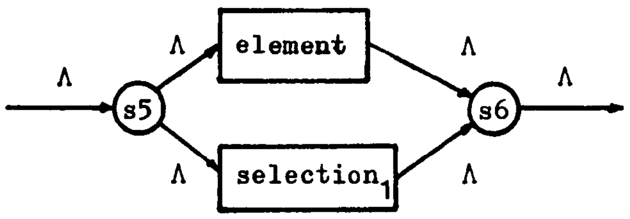
the final states are those of 'element$_2$'.

T3b: unit ::= selection



The initial and final states are those of 'selection'.

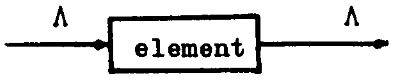
T4a: selection ::= element , selection_1



The initial state is s5 ∈ C2,

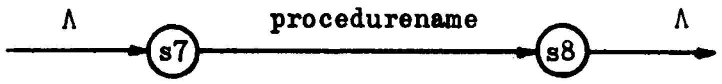
the final state is s6 ∈ C1.

T4b: selection ::= element



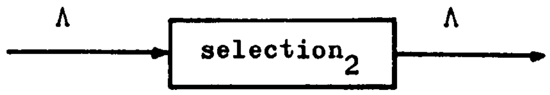
The initial and final states are those of 'element'.

T5a: element ::= procedurename



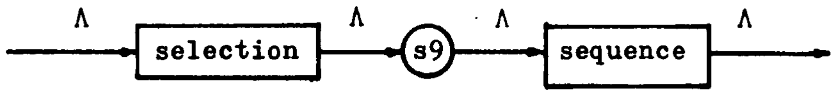
Since 'procedurename' is a terminal, it is the name of a transition. Hence, the initial and final states are s7 ∈ C2, and s8 ∈ C1, respectively.

T5b: element ::= (selection_2)



The initial and final states are those of 'selection_2'.

T5c: element ::= (selection ; sequence)



where s9 ∈ C1 and C2.

The initial states are those of 'selection',

the final states are those of 'sequence'.

First, we shall show that the transition system for 'element' has a single initial state belonging to C2, and a single final state belonging to C1.

For the initial states of 'element' and 'selection' in the transition systems T5 and T4 we have:-

T5a: 'element' has a single initial state, say s7, and s7 $\in$ C2.

T4a: 'selection' has a single initial state, say s5, and s5 $\in$ C2.

Suppose, as induction hypothesis, that 'selection' has a single initial state which belongs to C2. Then, we have:-

T5b and T5c: 'element' has the initial states of 'selection', and hence has a single initial state belonging to C2.

T4b: 'selection' has the initial states of 'element', and thus has a single initial state belonging to C2.

Therefore, by induction, we conclude that 'selection' in the transition system T4 has a single initial state belonging to C2, and also that 'element' in the transition system T5 has a single initial state belonging to C2.

1. Consider the final states of 'sequence' in the transition system T2. We have:-

T2b: 'sequence' has the final states of 'unit'.

Suppose, as induction hypothesis, that 'sequence' has the final states of 'unit'. Then, we have:-

T2a: 'sequence' has the final states of 'sequence_1'.

Therefore, by induction, 'sequence' has the final states of 'unit'.

For the final states of 'element' and 'selection' in the transition systems T5 and T4 we have:-

T5a: 'element' has a single final state, say s8, and s8 $\in$ C1.

T4a: 'selection' has a single final state, say s6, and s6 $\in$ C1.

T4b: 'selection' has the final states of 'element'.

2. The transition system for 'unit' is such that:-

T3a: 'unit' has the final states of 'element'.

T3b: 'unit' has the final states of 'selection'.

Thus, by T4a and T4b 'unit' has either a single final state which belongs to C1, or the set of final states of 'element'.

3. Substituting the results of 2. in 1., 'sequence' has either a single final state which belongs to C1, or the set of final states of 'element'.

Suppose, as induction hypothesis, that 'element' has a single final state belonging to C1. Then, we have:-

T4a and T4b: 'selection' has a single final state belonging to C1.

3.: 'sequence' has a single final state belonging to C1.

T5b: 'element' has the final states of 'selection', and hence has a single final state belonging to C1.

T5c: 'element' has the final states of 'sequence', and hence has a single final state belonging to C1.

Therefore, by induction, we conclude that 'element' has a single final state belonging to C1, and hence also that 'selection', 'sequence', and 'unit' each have one single final state belonging to C1.

The remainder of the proof uses the above results to show that the spontaneous transitions of the system for α may be removed.

Consider T5c. Let f be the final state of the transition system for 'selection', and a" be the initial state of the transition system for 'sequence'. Then, in the transition system T for 'element':-

$P(f) = s9$, and $f \in C1$.

Hence, E1 applies, the spontaneous transition may be removed, and s9 combined with f to give a new state $\{f \cup s9\}$. Again, we have:-

$P(\{f \cup s9\}) = a''$ and $\{f \cup s9\} \in C1$.

Thus, we may combine $\{f \cup s9\}$ with a" to form $\{f \cup s9 \cup a''\}$ and eliminate the spontaneous transitions between 'selection' and 'sequence'.

Consider T4a. Let a and f be the initial and final states of 'element' respectively. Let a" and f" be the initial and final states of 'selection'. Then in the transition system T for 'selection' we have:-

$P(s5) = a$ and $a \in C2$. Thus, by E2 we may combine s5 and a.

$P(\{s5 \cup a\}) = a''$ and $\{s5 \cup a\} \in C2$. Thus, by E2, we may combine $\{s5 \cup a\}$ and a" to give $\{s5 \cup a \cup a''\}$. We note that the initial state of 'selection' is still, after reduction, a member of C2. In addition, we have:-

$P(f) = s6$ and $f \in C1$. Thus, by E1, we may combine s6 and f.

$P(f'') = \{s6 \cup f\}$ and $f'' \in C1$. Thus, by E1, we may combine $\{s6 \cup f\}$ and f" to form $\{s6 \cup f \cup f''\}$. We note that the final state of 'selection' is still, after reduction, a member of C1. Thus, all the spontaneous transitions introduced by T4 may be removed.

Consider T3a. Let a and f be the initial and final states of 'element_1' and a" and f" be the initial and final states of 'element_2'. Then in the transition system T for 'unit' we have:-

$P(s3) = a$ and $a \in C2$. Thus, by E2, we may combine s3 and a.

$P(f) = \{s3 \cup a\}$ and $f \in C1$. Hence, by E1, we may combine $\{s3 \cup a\}$ and f.

$P(\{s3 \cup a \cup f\}) = s4$ and $s4 \in C2$. Hence, by E2, we may combine $\{s3 \cup a \cup f\}$ and s4.

$P(\{s3 \cup a \cup f \cup s4\}) = a''$ and $a'' \in C1$. Thus, by E1, we may combine $\{s3 \cup a \cup f \cup s4\}$ and a" to form $\{s3 \cup a \cup f \cup s4 \cup a''\}$.
Note that there is only one initial state of 'unit' after reduction.
Thus, all the spontaneous transitions introduced by T3 may be removed.

Consider T2a. Let f be the final state of 'unit', and a" be the initial state of 'sequence$_1$'. Then for the transition system T for 'sequence':-

$P(f) = s2$ and $f \in C1$. Hence, by E1, we may combine f and s2.

$P(\{f \cup s2\}) = a''$ and $\{f \cup s2\} \in C1$. Hence, by E1, we may combine $\{f \cup s2\}$ and a".
Thus, all the spontaneous transitions introduced by T2 may be removed.

Consider T1. Let a and f be the initial and final states of 'sequence'. Then in the transition system T for 'E-path':-

$P(s1) = f$ and $f \in C1$. Hence, by E1, we may combine s1 and f. The initial states of 'sequence' are those of 'unit'. However, even after reduction, 'unit' has a single initial state.

$P(a) = \{s1 \cup f\}$ and $\{s1 \cup f\} \in C1$. Hence, by E1, we may combine $\{s1 \cup f\}$ and a.
Thus, all the spontaneous transitions introduced by T1 may be removed.

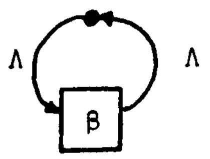
'Procedurename' contains no spontaneous transitions. Suppose, as induction hypothesis, that the regular expression β, contained in α, has a transition system which contains only spontaneous transitions which may be removed. We may apply one of the E-path productions and its corresponding transition system construction rule to β to obtain γ and a transition system for γ. However, all the spontaneous transitions introduced in forming the new transition system from that of β may be removed. Thus, all the spontaneous transitions in the transition system for γ may be removed. Therefore, by induction, we conclude that all the spontaneous transitions in the transition system for α given by 2.19 may be removed.

Q.E.D.

As a corollary, the construction 2.19 can be simplified to produce a finite state machine directly from the E-path expression without producing an intermediate transition system. In addition, 3.1 implies the following property:-

Theorem 3.2: Let α be a regular expression such that path α end is an E-path expression. Then the non-deterministic finite state machine constructed by 3.1 is strongly connected.

Proof: Application of 2) of 2.19 to α gives the transition system:-



where α is the regular expression (β)*..

This transition system is strongly connected (2.21).

Each of 2), 3) and 4) preserves the strongly connected property of the graph and introduces new vertices which are also strongly connected by having directed arcs to and from other strongly connected vertices.

Each of the rules E1 and E2 of 3.1 preserves the strongly connected property of the graph since it only combines vertices.

Q.E.D.

This theorem implies that there must always be a sequence of executions of the procedures in an E-path expression which will lead to a given procedure being capable of being executed by some process. Of course, this gives no assurance that the processes are themselves deadlock free in the manner in which they invoke procedures of the paths, or that the combination of several paths in a program will not give rise to deadlock.

The following lemma will be used in a proof that the E-path notation can be implemented using P and V operations in the prologues and epilogues of procedures.

Lemma 3.3: Let α be a regular expression such that path α end is an E-path expression. Then the state machine (U) constructed for α by 3.1 has no two transitions which share the same name, is deterministic, and is minimal (2.18).

Proof: Each of the four construction rules of 2.19 generates transition systems which have distinct parts corresponding to distinct subexpressions of the original regular expression. Thus, if two transitions have the same name (excluding spontaneous transitions), then they must have been generated from two identical regular expressions. However, if the original expression α contained two such expressions, then it would contravene restriction R1 and path α end would not be a valid E-path expression. Hence, the transition system for α cannot have two transitions which share the same name. 3.1 removes transitions which are spontaneous but does not add further transitions. Hence, the resulting state machine U has no two transitions which share the same name.

Consider the state machine U. If e is a member of the alphabet of the state machine and s is one of its states, then the transition function with arguments e and s can only be single valued since no two transitions have the same name. Hence, by definition 2.13 and 2.14 the state machine is deterministic.

Finally, assume that the state machine U is not minimal. Then, the minimal state machine is obtained by finding the equivalence classes of U (Myhill, 1957). There must exist at least two states u and v of the state machine U which belong to the same equivalence class. However, suppose that a transition from u accepts the first member e of the sequence z of letters belonging to the alphabet of the state machine. (There must be at least one such transition from u since the state machine is strongly connected (3.2).) Then e must also be accepted by a transition from v. But this contradicts the result that the state machine U has no two transitions which share the same name and hence that the state machine was constructed by 3.1. Thus, the state machine constructed by 3.1 is minimal.
Q.E.D.

3.5 STATE MACHINE CONTROLLERS AND A P, V OPERATION IMPLEMENTATION.

One of the important properties that we require for the E-path notation is that it should be readily implementable on the majority of present day machines. In Chapter 1 we illustrated an implementation in terms of counting semaphores for that form of path notation. In this chapter we establish an implementation for E-paths in terms of binary semaphores (Shaw,1974).

To implement an E-path expression, we associate with each state in the equivalent state machine constructed by 3.1 a binary semaphore which will have the value one whenever the machine is in that state and will have the value zero otherwise. For each synchronized action (represented by a transition in the state machine), we provide the corresponding procedure with a prologue containing a P operation on the semaphore representing the state from which the transition takes place. The procedure is also provided with an epilogue containing a V operation on the semaphore representing the state to which the transition occurs. Thus, for example, we have for an action P:-

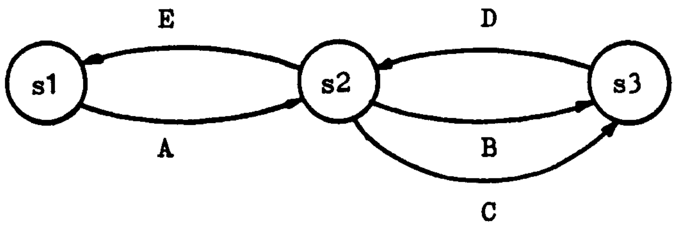
<u>procedure</u> P: <u>begin</u> P(s_i) ; (body of procedure P) ; V(s_j) <u>end</u>

where s_i and s_j are semaphores whose values will be used to encode the state of the state machine. Execution of the procedure by a process corresponds to a change of state in terms of the state machine.

For example, the following E-path:-

<u>path</u> (A ; ((B , C) ; D)* ; E)* <u>end</u>

has the corresponding state machine:-



which is implemented as:-

```
semaphore s1 = 1;  semaphore s2, s3 = 0;
procedure A:  begin  P(s1);  (body of A);  V(s2)  end;
procedure B:  begin  P(s2);  (body of B);  V(s3)  end;
procedure C:  begin  P(s2);  (body of C);  V(s3)  end;
procedure D:  begin  P(s3);  (body of D);  V(s2)  end;
procedure E:  begin  P(s2);  (body of E);  V(s1)  end;
```

(In Chapter 4 we shall give a meaning to the interaction of a path expression with the procedure calls of a program using a Petri net representation. This will permit us to represent mathematically the interaction between processes and path expressions. In particular, we shall show that the above implementation is correct with respect to this representation, hence permitting the theoretical results obtained from the representation to be applied in practice.)

The E-path expressions are based upon expressing synchronization using a restricted regular expression. These restrictions are necessary to allow E-path expressions to be implemented using the scheme above. For example, the following path expression:-

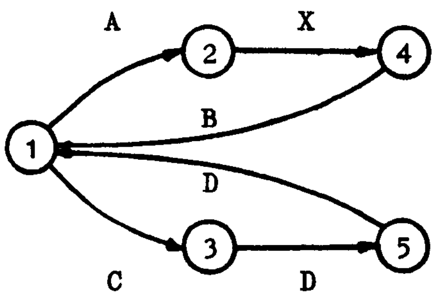
```
path ( ( A ; X ; B ) , ( C ; X ; D ) )* end
```

has no implementation of the form:-

```
procedure X;  begin  (mixture of only P, V operations) ;
                     (body of procedure X);
                     (mixture of only P, V operations)
                     end
```

The above path expression has the state machine representation (if we allow the repetition of procedure names):-

initial state



A process executing any of the procedures X, B, and D can only wait on one semaphore at a time. Associating semaphores with states, this implies we must combine states 2 and 3 together. However, states 2 and 3 cannot be combined without altering the behaviour of the state machine. Hence, in order to implement this path, we require more than just a simple association of states with semaphores. For example, we could introduce variables and array indexing or conditional statements (see Chapter 5). (This is similar to the well known problem of the cigarette smokers (Patil, 1971), (Parnas, 1975), (Habermann, 1972).) 1

Theorem 3.4: The restrictions imposed by the E-path notation on the set of regular expressions of which an E-path expression can be composed are sufficient to allow all such expressions to be represented by an implementation of the form:-

procedure X: begin $P(s_i)$; (body of procedure X); $V(s_j)$ end

Proof: As a corollary to 3.3, the finite state machine constructed for an E-path expression by 3.1 has only one state in which a particular procedure execution may be accepted. Hence this state can be represented by a semaphore upon which processes requesting the execution of that procedure may wait. Again, as a corollary, given a particular procedure execution, there is only one state into which the state machine may enter. This allows processes terminating their execution of that procedure to execute a signal to the processes waiting upon the semaphore which represents that state. Finally, the state of the state machine is represented by the semaphore corresponding to that state having the value one, all other semaphores having the value zero. Hence, the restrictions are sufficient. Q.E.D.

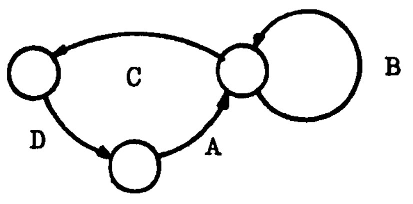
3.6 DECLARATIVE POWER

There are two measurements we shall make for a given path notation P1 which will be used to compare the ability of that notation to express synchronization with the ability of another path notation, say P2. These measurements are the declarative power and scheduling power of P1, written D(P1) and SP(P1) respectively. (SP(P1) will be defined in 4.6.) The comparison of D(P1) with that of D(P2) will be used to determine if P1 is better than P2 at meeting the goal of removing synchronization from program text and expressing it as a static declaration.

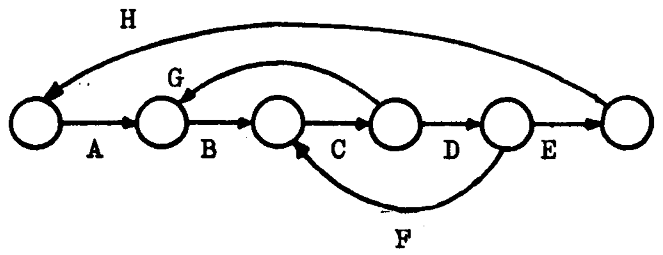
Let Si be the set of (possibly infinite) sequences of actions accepted by a path expression Ei. (We will define Si to be the set of sequences generated by the simulating net of Ei, see simulating nets in 4.1.) D(P1) is defined to be the set of elements Si such that Ei is an expression of P1. D(P1) is at least D(P2) if D(P1) contains D(P2). (That is, for every path expression E2 of P2 there is an E1 of P1 such that S2 is accepted by E1.) If the elements of D(P1) are the same as those of D(P2) then D(P1) equals D(P2). If D(P1) contains D(P2) but there are elements of D(P1) not in D(P2) then D(P1) is greater than D(P2).

We may obtain an informal characterization of the declarative power of E-path expressions by identifying the properties of the finite state machines to which they are equivalent. The state machine corresponding to an E-path has the following properties:-

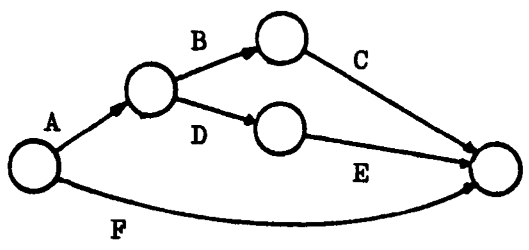
1) It is strongly connected (3.2).
2) It only has one state transition to represent any procedure execution (an equivalent statement of 3.3).
3) It has no non-nested loops. That is, E-paths may generate loops of the form:-



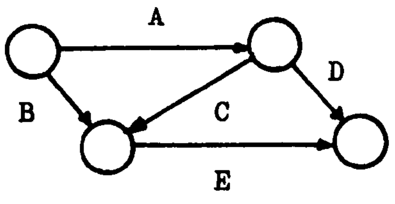
where the loops are properly nested, but not of the form:-



4) It has no non-nested selections. That is, E-paths may generate selections of the form:-



where the selections are nested, but not of the form:-



5) It allows no parallelism. (This follows directly from 1) and amounts to an E-path corresponding to one state machine, not several independent ones.)

The potential power of expression of synchronization belonging to the implementation of the E-paths may be identified as equivalent to the expressive power of state machines in which no two transitions share the same name. This includes state machines with properties such as non-nested loops and selections as in 3) and 4) above. Thus, to conclude the discussion of the E-path notation we shall briefly consider other notations which would exploit further this implementation scheme to improve upon the declarative power of E-path expressions.

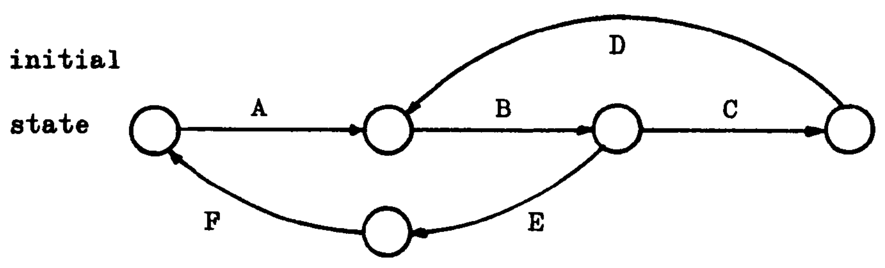
3.7 REMARKS ON MODIFICATIONS TO THE NOTATION.

We will only consider modifications which would take advantage of the implementation scheme described above to increase the declarative power of the notation. In addition, we shall require the modifications to allow expressions of the notation to be checked syntactically to determine that they may be implemented using the desired mechanism, Thus, for example, we shall not consider permitting any regular expression to be used within an E-path expression. Such expressions would require more than a trivial syntactic analysis before a decision could be made that they were implementable (for instance, by minimizing the state machine given by 2.19 corresponding to the regular expression). We will demonstrate that, having such a fixed implementation scheme, increasing the power of declaration of the path notation is closely linked with the structure we wish to impose on synchronization specifications. We will describe two modifications which are successively more powerful and which, as a result, lose structure.

The first modification replaces the Kleene star by a loop and exit mechanism. The loop does not terminate until an exit is taken. The modified notation allows some non-nested loops and selections to be expressed. For example, we may have:-

<u>path</u> <u>loop</u> (A ; <u>loop</u> (B ; (C , E <u>exit</u>) ; D) ; F) <u>end</u>

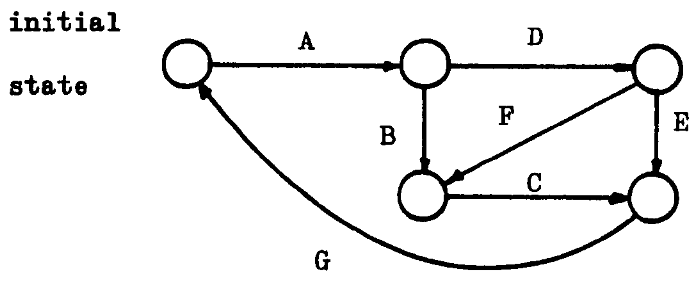
which corresponds to the state machine:-



or we may have:-

<u>path</u> <u>loop</u> (A ; <u>loop</u> ((B,(D ; (E <u>exit</u> , F))) ; C <u>exit</u>) ; G)<u>end</u>

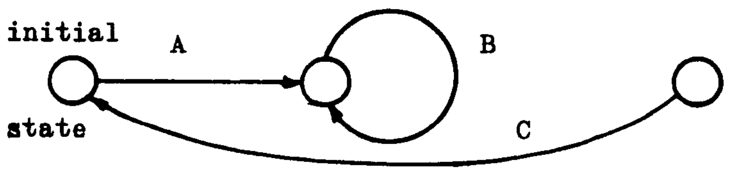
which has the state machine representation:-



However, the modified notation allows expressions to be written which are not strongly connected. For example:-

<u>path</u> <u>loop</u> (A ; <u>loop</u> (B) ; C) <u>end</u>

which has the state machine representation:-



where C may never occur and A may occur once only.

The second modification attaches a label to any sequence. Each procedure name in the path expression may be followed by a goto. This notation allows the full potential of the implementation to be utilized. The problem we have outlined above is one of language design, a trade-off between the power of description, understandability, and possibility of error. We shall not concern ourselves further with the solution to this problem.

3.8 A SUMMARY OF E-PATH EXPRESSIONS.

We have defined a path notation by giving a set of productions which generate a synchronization expression based on regular expressions. Each path expression could be represented by a finite state machine. The notation was constructed so that path expressions would have corresponding state machines with properties which would allow them to be easily implemented using P and V operations on binary semaphores. Finally, we gave two possible ways in which the power of synchronization of a path notation could be characterized, and discussed the E-path notation with respect to one of them.

CHAPTER 4

PROCESSES AND PATH EXPRESSIONS.

The elementary path notation is defined by a set of productions and a meaning is attached to an expression of the notation by relating it to a state machine or regular expression. Such a state machine or regular expression accepts an alphabet composed of the procedure names which are synchronized by the path expression. The path expression permits the execution by processes of these procedures in the same sequence as the corresponding state machine would accept the names of the procedures. In this chapter we shall provide a more formal description of the relationship between path expressions and processes.

The finite state machine corresponding to an elementary path expression can be represented as a State Machine Petri net (2.9). In addition, a general process can be represented by a Petri net of the appropriate kind, see (Petri, 1962). We shall introduce a Petri net model for programs consisting of systems of processes and path expressions to provide a framework for the study of the impact of process structuring and path structuring on the synchronization properties of programs. We can then demonstrate some of the more obvious results from our model:-

a) The correctness of the implementation of path expressions. (In later chapters we shall demonstrate the correctness of implementations of path expressions belonging to other path notations using the same technique.)

b) A classification of processes based upon the model. For one such class, together with a subclass of the E-path notation, there exists a simple criterion which guarantees freedom from deadlock.

c) The structuring properties that a type definition imposes on the representation of processes and paths, and the usefulness of separating the specification of an object from its implementation.

d) Wherever possible, how a general study could be organized to investigate the synchronization properties of programs described using that model.

The model of a program is presented first, and this will be used to describe the relationship between the E-path notation and processes. We will then present some of the results from our model.

4.1 PATH EXPRESSIONS AND SIMULATING NETS.

We shall use Petri nets in our model of a program to represent E-paths and to define their meaning. By the use of Petri nets we can provide a machine independent definition of the semantic meaning of the E-paths. The problems of scheduling which frequently arise to confuse the issue of synchronization can be left entirely arbitrary. (The effects of different scheduling algorithms on synchronization primitives have been a source of much discussion, for example see (Lipton, 1973).) The description of E-path expressions will involve only synchronization, allowing a greater freedom in the way these paths may be implemented. This is reflected in the Petri net representation. (For example, the work of Patil (Patil, 1975) and others has shown that such Petri nets can be realized with electronic circuits, hence permitting hardware implementations of E-paths.) In addition, Petri nets provide a mathematical tool (Lautenbach and Schmid, 1974) about which many theorems and classifications are known or being developed. Hence they provide a convenient tool with which to analyze path expressions and their interactions with processes.

A Petri net whose transitions are labelled with the names of the procedures in a path expression is the <u>simulating net of that path expression</u> if and only if it generates the sequence of actions which are accepted by the path expression.
(A sequence is generated by a labelled Petri net by repeatedly firing an enabled transition and recording its label as the next member of the sequence.) We shall call simulating nets of E-paths <u>E-nets</u> for short.

Translation of an E-path expression into a corresponding E-net is accomplished trivially by using the state machine constructed by 3.1 to specify the corresponding State Machine Petri net (2.7). The initial state of the state machine is represented by marking the corresponding place of the net with a token. The following lemma follows directly:-

<u>Lemma 4.1</u>: E-nets are live and safe.

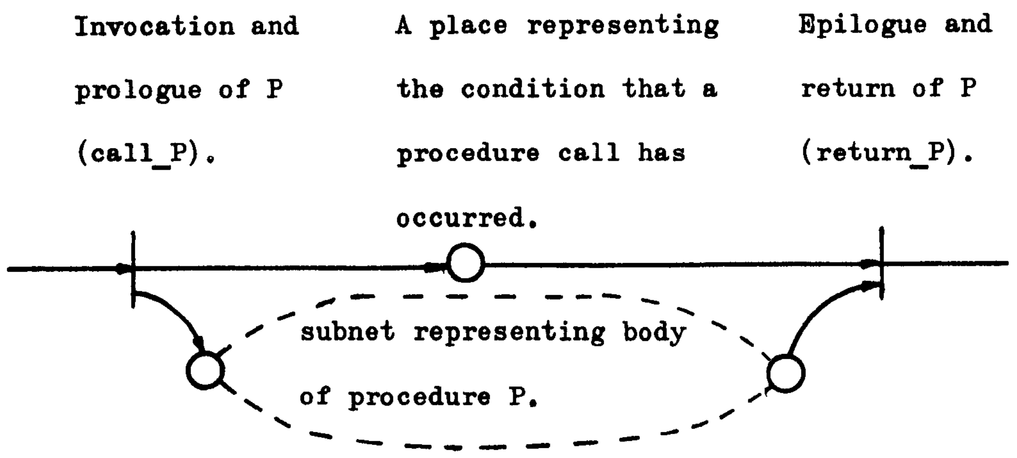
<u>Proof</u>: From 3.2 the state machine net will be strongly connected (2.8), and by construction it will have one marker. Hence, by 2.11 it is live and safe.

In the present context, the properties of liveness and safeness add nothing further to the understanding of E-path expressions. However, they will become important later when we consider problems of deadlock and the possibility of simultaneous execution of a given procedure.

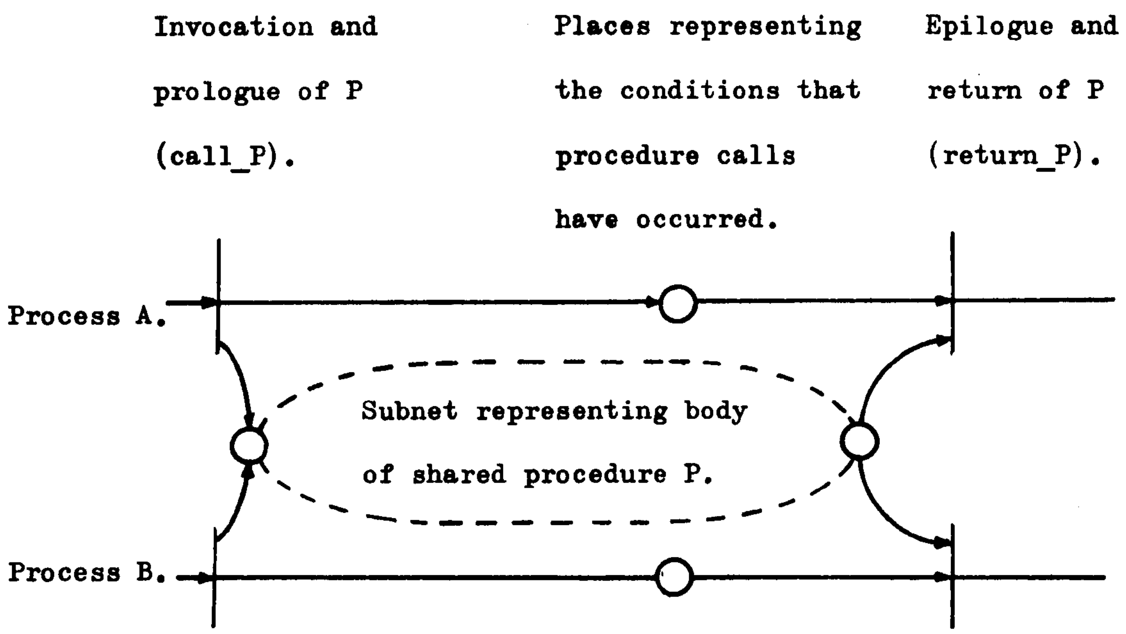
4.2 AN ABSTRACT MODEL OF A PROCESS.

An abstract model of a process, for the purposes of this thesis, consists of a characterization of a process by means of a marked Petri net (called the simulating net of the process) which generates a set of sequences of actions that model the execution of the code of the process. The dynamic behaviour of the process is represented by the movement of tokens through the net as the transitions fire. In a similar fashion to the simulating nets of E-paths, transitions will be used to represent actions (that is, occurrences of events) and will be labelled by the piece of code that they represent. Such models have been used by Patil (Patil, 1971), Schmid (Schmid, 1974), Lautenbach (Lautenbach, 1974), and others. This model concentrates on representing the 'control structure' of a process and the notion of data and value are abstracted.

In general, each process of a system is represented by a separate simulating net. However, if processes can communicate with each other, this will be represented by connecting their corresponding nets with arcs and intermediate places. Such an intermediate place will both be an output place of a transition belonging to the process initiating the communication, and an input place of a transition in the process receiving that communication. For more complex forms of interprocess communication we allow the use of shared procedures. The body of the procedure is represented by a single subnet with one input place and one output place. Each call of the procedure in each process is represented by a transition representing the invocation and prologue of the procedure, a place to mark where the call came from, and a transition representing the epilogue and return of the procedure. For example, a procedure P which is invoked from only one point in a process would be represented by:-


Invocation and prologue of P (call_P).
A place representing the condition that a procedure call has occurred.
Epilogue and return of P (return_P).
subnet representing body of procedure P.

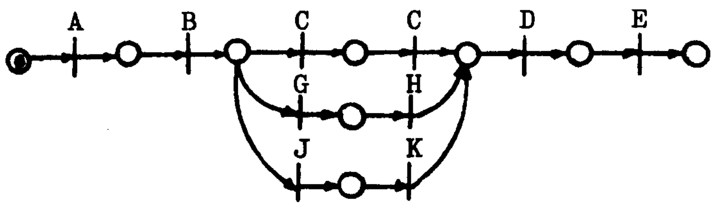
Suppose that P can be called from two points in separate processes. Then we would represent P by:-


Invocation and prologue of P (call_P).
Places representing the conditions that procedure calls have occurred.
Epilogue and return of P (return_P).
Process A.
Subnet representing body of shared procedure P.
Process B.

(Note that if both processes invoke P at approximately the same moment the input places to P will not be safe. However, we may be able to assert that it is n-safe for some n greater than zero (2.4, 2.5). Hence, we may use this characterization to investigate whether a given procedure may be executed simultaneously by several processes by using the notion of safeness and appropriate theorems from Petri net theory.)

In principle, our model ought to represent every atomic action executable by each process. However, this would lead to extremely cumbersome simulating nets and would obscure our real purpose. Therefore we will introduce abbreviations which will allow us to collapse whole subnets into single transitions when this does not affect our results. To avoid the general problem of deciding whether or not a particular abbreviation can be made (Lauer, 1972), we shall restrict their use and form. Abbreviations will be used to remove unwanted detail from the discussions on synchronization, while the form of these abbreviations will be restricted to ensure the retention of those properties of the program we wish to investigate. In general, we shall restrict abbreviations to substituting single transitions for subnets which correspond to subsequences of some of the sequences of actions generated by a simulating net.

Hence, for example, suppose we have the simulating net:-



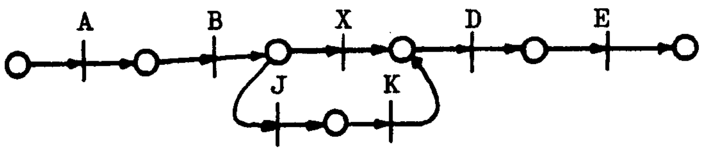
which generates the set of sequences:-

```
        A  B  C  C  D  E
or      A  B  G  H  D  E
or      A  B  J  K  D  E
```

then subsequences of some of those sequences are:-

```
              C  C
or            G  H.
```

These may be abbreviated to a single transition X, the resulting net being:-

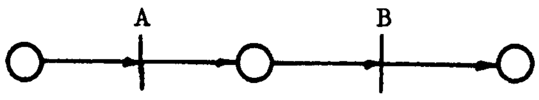


which generates the set of sequences:-

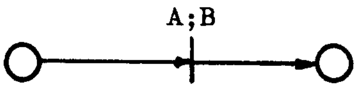
```
        A  B  X  D  E
or      A  B  J  K  D  E
```

In particular, the form of the abbreviations will be as follows:-

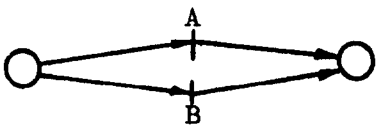
a) For a subnet representing a sequence we can abbreviate the sequence from the form:-



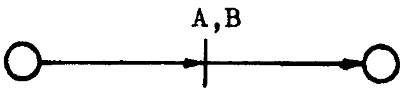
to:-



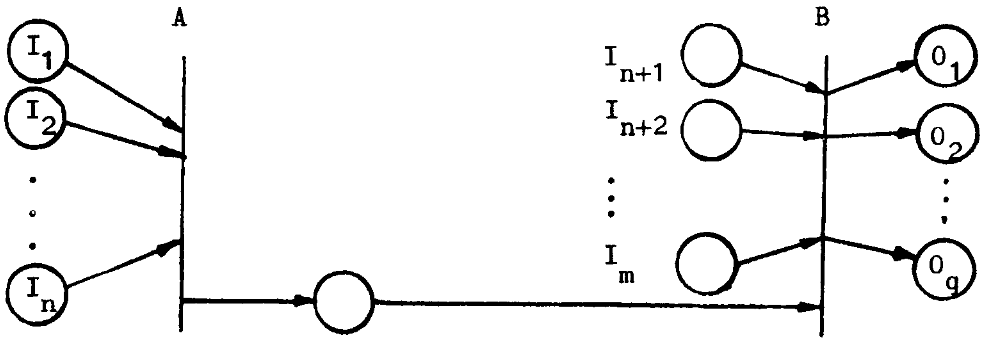
b) For a subnet of the form:-



we can substitute the abbreviation:-

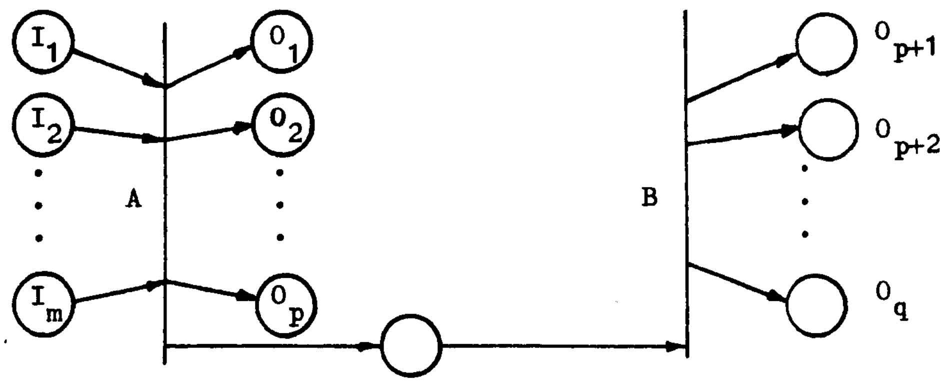


c) For a subnet of the form:-

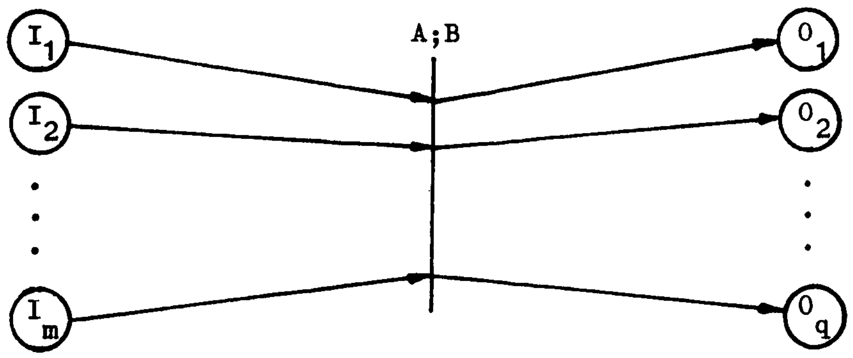


where I_1-I_n are input places only of transition A,

or for a subnet of the form:-

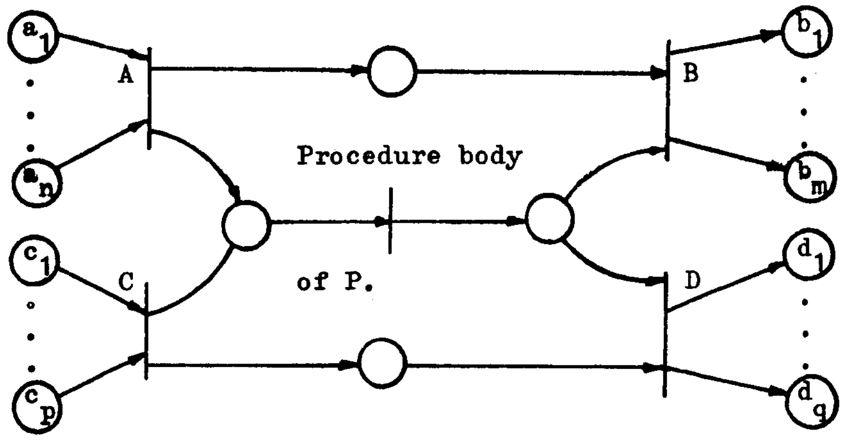


we can substitute the following abbreviation:-

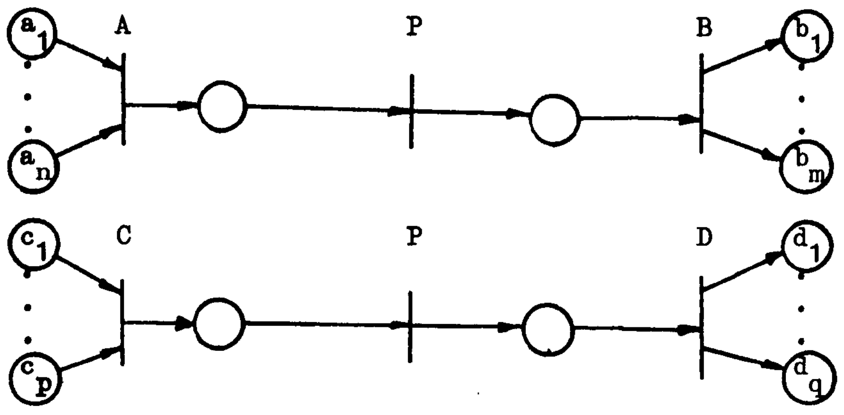


for all values of m, p and q such that $0 \leq n \leq m$ and $0 \leq p \leq q$.

d) For a subnet of the form of a procedure call:-



we can abbreviate the body and calls of the procedure by a set of single transitions, provided that we are not interested in investigating whether that procedure may be executed simultaneously by several processes:-



where n, m, p and q ≥ 1.

In the unabbreviated net, P may be enabled by the firing of A or C. The abbreviation is to replace the single transition for the procedure P by a set of transitions all representing P. A transition labelled P can fire, in the abbreviation, whenever the single transition labelled P in the original net can fire.

<u>Lemma 4.2</u>: The abbreviations a), b), c) and d) leave the behaviour of the surrounding simulating nets unchanged.

<u>Proof</u>: For a) and b) the proof is trivial. For c) we note that there is no way in which the surrounding net can distinguish between the firing of an A and the firing of a B. In the first subnet form of c), the surrounding net having deposited tokens on each of I_1-I_n cannot remove them since these places are only input places to transition A. Hence, it cannot detect an A occurring until a B has occurred. In the second subnet form, the occurrence of an A implies the occurrence of a B and hence is independent from the surrounding net.

Finally, for d), if A fires in the original net, first P and then B must eventually fire, marking places at the ends of the arcs b_1-b_m. The actual occurrences of A, P or B are indistinguishable from each other with respect to the surrounding net. Hence, making the abbreviation does not alter the behaviour of the surrounding net. Q.E.D.

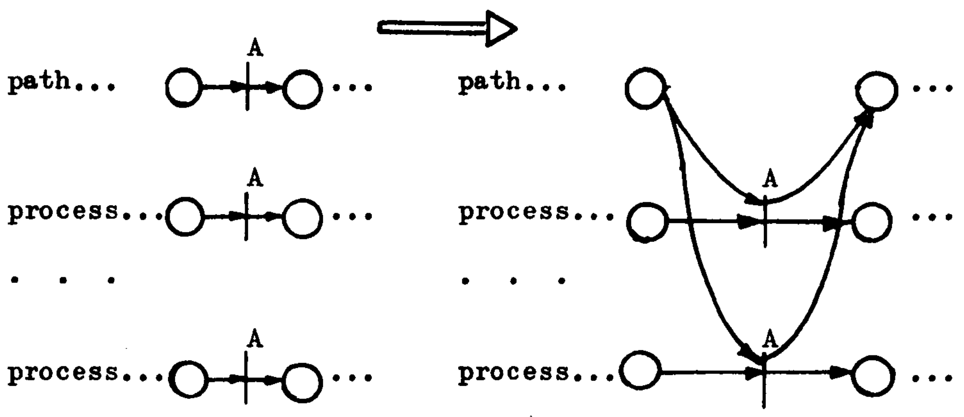
4.3 A MODEL FOR A PROGRAM CONSISTING OF PATHS AND PROCESSES.

The model of a process may be combined with the Petri net representation of E-paths given in Chapter 2 to provide a model for programs. A program may include several processes and path expressions. The individual Petri net representations of these paths and processes may be combined by identifying individual actions in the paths and processes and ensuring that they may only occur when both the set of processes requests those actions, and the set of paths permits those actions to occur. The following intersection transformation gives the required combination:-

TR4.3 Intersection transformation for processes and E-paths.

1) Abbreviate all calls upon shared procedures using 4.2 d).

2) A given action or procedure name will occur once only in a set of E-path expressions as a result of the restriction on the repetition of names within and between E-paths given in Chapter 3. Hence, for each labelled transition in the simulating nets of the path expressions there will be a set of transitions labelled with an identical name in the simulating nets of the processes. Perform the following transformation on such transitions:-



The initial marking of the net is left unchanged by the transformation.

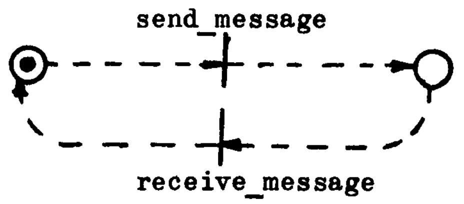
The firing of a transition in the simulating net of the process is now made to be dependent on the appropriate condition holding in the simulating net of the path. This construction places the synchronization around the procedure call mechanism. (This may be seen by expanding the shared procedure abbreviation for A in the above transformation.) Hence, we may examine the safeness of the input and output places of a shared procedure body to determine whether it is in a 'critical section' or may be executed simultaneously by several processes.

Example of a program consisting of paths and processes.

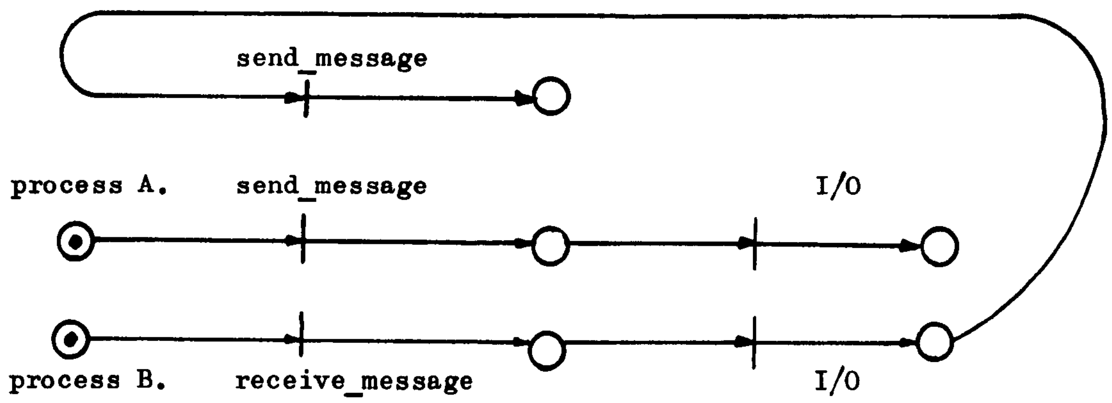
Two processes are communicating with each other by sending messages through a buffer. An E-path synchronizes the use of the buffer so that the action of sending a message must be followed by the action of receiving a message:-

path (send_message ; receive_message)* end

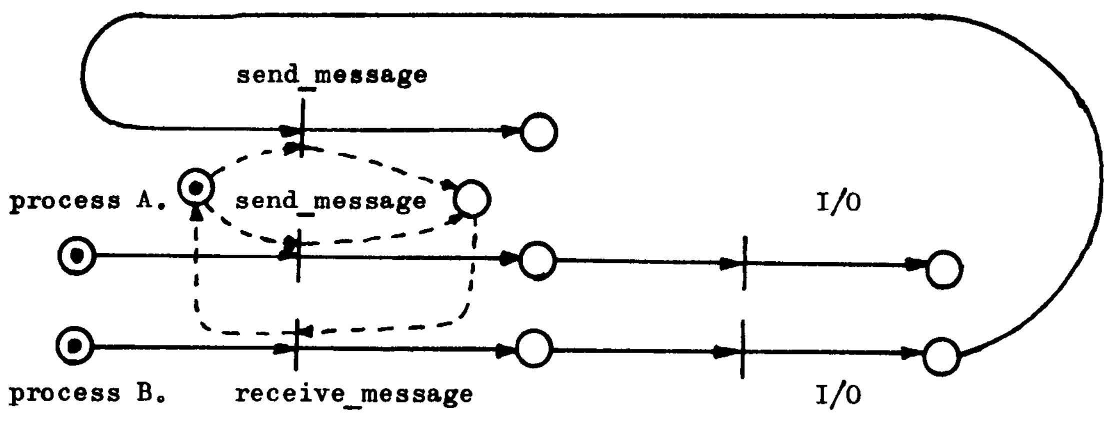
This has the following simulating net:- (In this and later examples, to illustrate the component subnets of a Petri net, we will draw some of the arcs with dotted rather than solid lines. Of course, the dotted lines have no role in any theoretical or analytical study of the nets. We shall use dotted arcs in the following to distinguish the subnet corresponding to the simulating net of the E-path.)



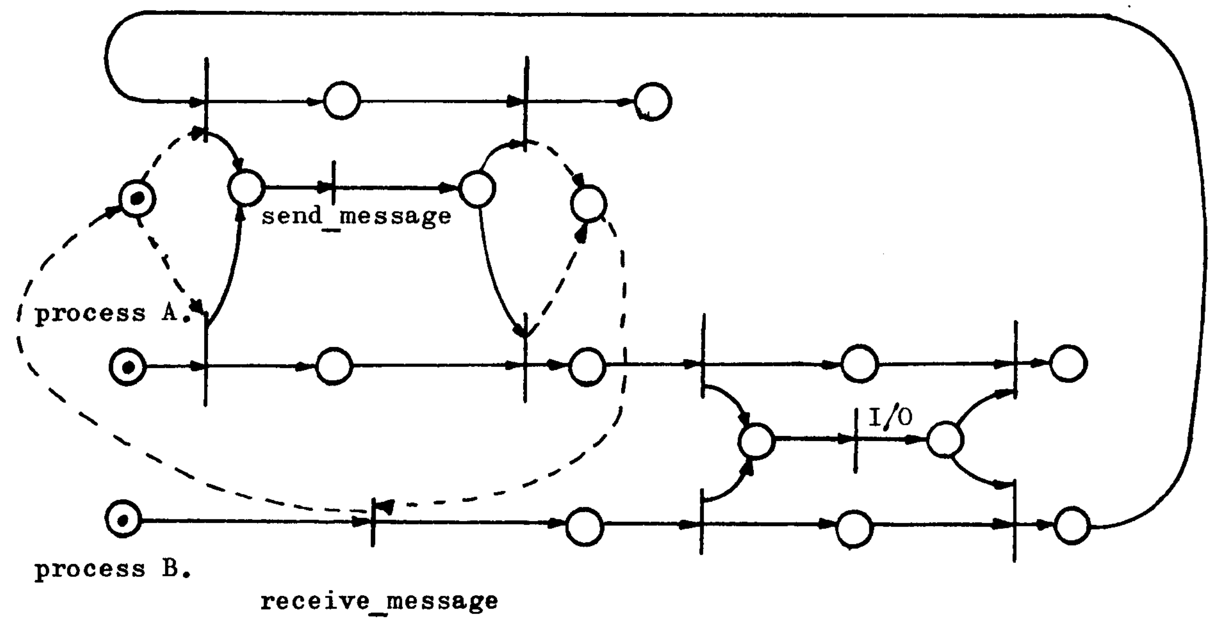
The simulating net model of the two processes is:-



where the shared procedures 'send_message' and 'I/O' have been abbreviated. Using the intersection transformation to combine the processes and paths yields:-



Finally, to examine whether the buffer may be accessed simultaneously by processes, we may expand the procedure call abbreviations and find whether the net is safe:-



The input and output place to send_message is one safe because of the surrounding path expression net. The input and output places of I/O are not one safe (for example, if send_message occurs followed by receive_message).

We have introduced a model of a program based on Petri nets. In the following section we shall investigate the structure imposed upon synchronization and co-ordination by type definitions, describing this structure by means of the model of a program.

4.4 TYPES.

To complete the model of a program we shall discuss a representation for types and objects created by instantiating types. This will serve two purposes:-

1) It will permit a program to be factored into logical units which may be represented independently.
2) It will allow an investigation into the structure resulting from associating path expressions with type definitions to describe shared objects.

The representation for a <u>type</u> will be a Petri net which is used as a definition for the construction of nets representing individual instances of that type (i.e., objects). As such, the representation of a type does not appear in a program, although instances of its representation will occur. The type may contain calls upon procedures not included in that type and hence a characterization of a particular type is considered with the surrounding program or environment in which it is defined.

An <u>instance</u> or <u>object</u> created from a type is a subnet of the program which is isomorphic (Harary,1969) to the net representing the type. The transition labels will be different; if the Simula dot notation is followed, then a transition labelled ⟨name⟩ in the type definition will appear as:-

⟨identifier_of_instance⟩ . ⟨name⟩

in the object.

(Note that we wish to use a Simula-like dot notation for its convenience, without entering into a discussion about the use and representation of reference variables. Hence we shall restrict the notation we use to meet our requirements and represent that notation in the simulating net in the simplest possible way.)

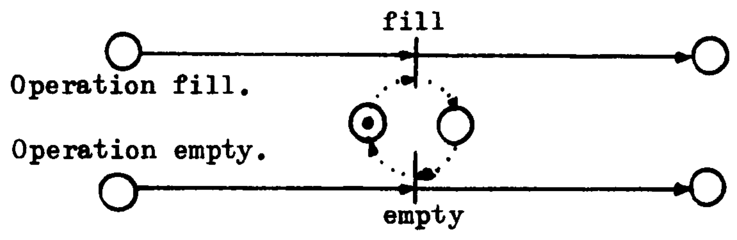
For the purposes of investigating program behaviour, the properties of a given type definition will recur in each of the subnets of the objects in the program. Hence, such investigations can be simplified to an examination of the connections of the subnets of each object to the surrounding program net and the way in which these connections affect the internal properties of the object. The number and manner of connections which can be made from an instance of a type to its surrounding program is restricted. In accordance with such programming languages as Pascal and its monitors, the restriction we impose is that all the connections made in this way must be done through a group of procedures called the operations (Campbell and Habermann, 1974) of the type (these are termed entry procedures in monitors). Hence, the connections will be made, in terms of the Petri net representation, as procedure calls.

<u>Example of the use of types in a program of path expressions and processes.</u>

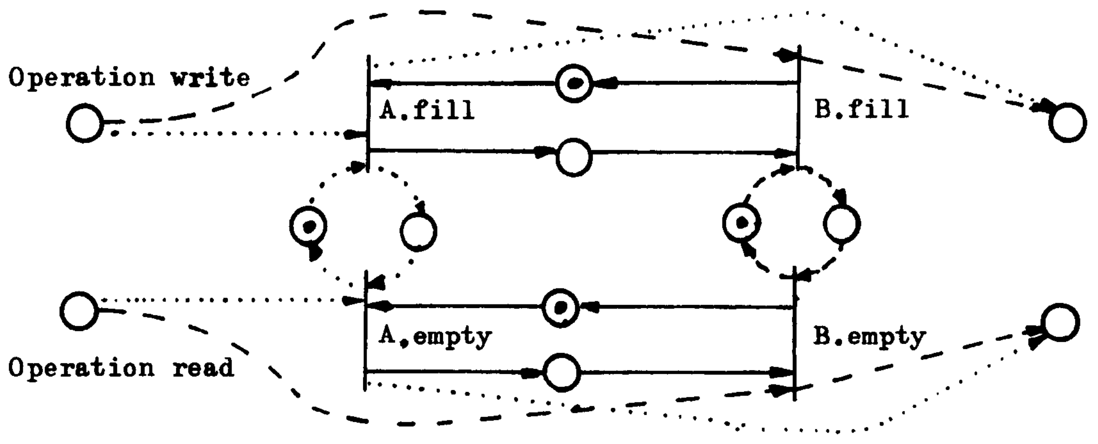
A compiler has been written to compile code using three passes corresponding to lexical analysis, semantic analysis and code production. Each pass of the compiler is implemented by a separate process, these processes communicating through a buffering system to permit concurrent operation. The lexical analysis produces output called 'tokens', the semantic analysis produces output called 'I-code' and the code production produces output called 'O-code'. Buffering is required for the temporary storage of the 'tokens' and the 'I-code', and is provided by an instance of type 'buffer'. Each instance of type 'buffer' has two 'frames' which are used to store information. These instances of 'frame' are used alternately, and one may be 'emptied' whilst the other is 'filled'. The type 'frame' has two operations, 'fill' and 'empty', and they are synchronized by the following E-path expression:-

<u>path</u> (fill ; empty)* <u>end</u>

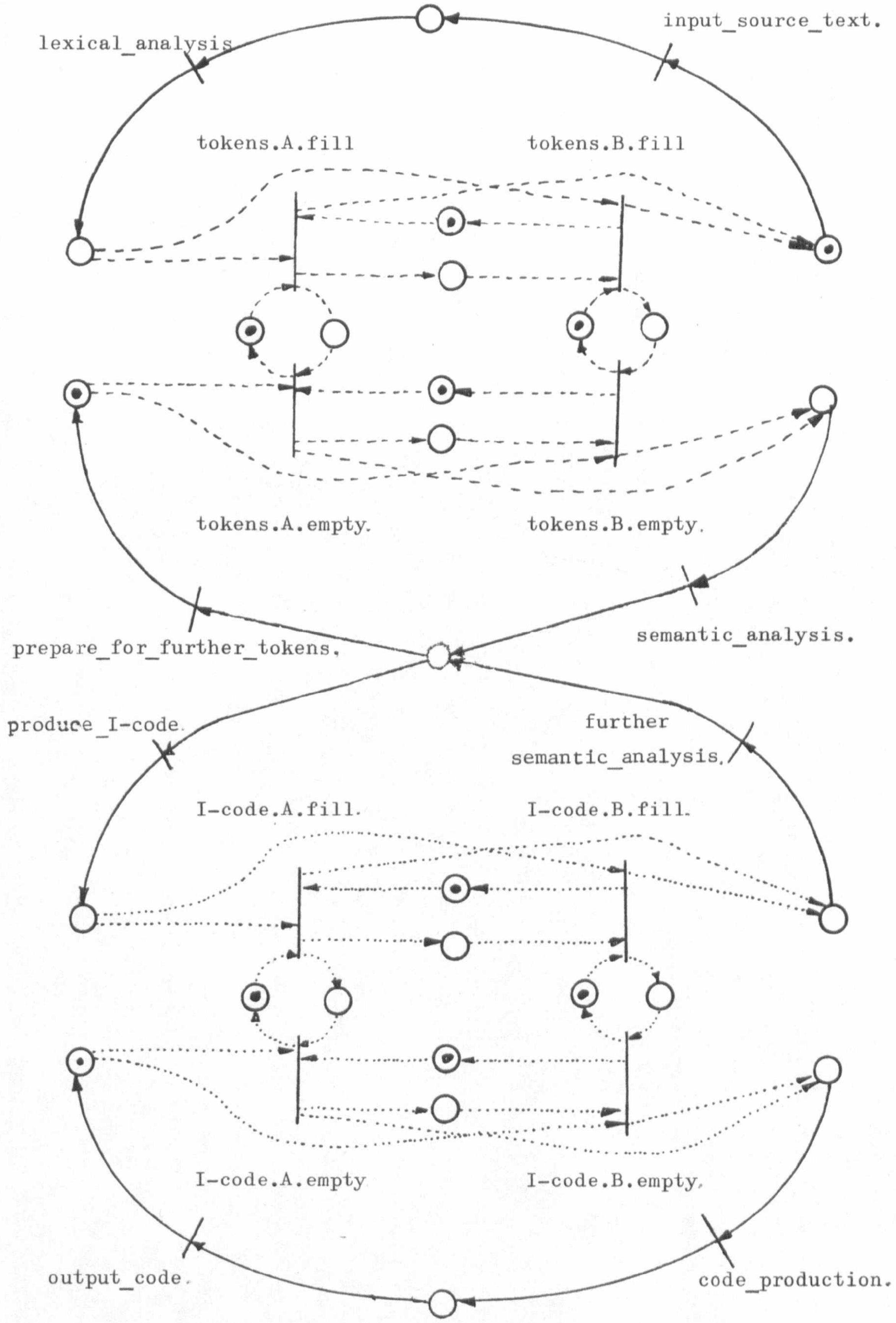
The type 'frame' can be described by the following net (the E-net corresponding to the E-path is shown with dotted arcs):-



The type 'buffer' has two operations 'read' and 'write'. The operation 'write' chooses between filling a 'frame' 'A' or a 'frame' 'B'. The choice is made alternately and is enforced by a subnet. The operation 'read' chooses between emptying a 'frame' 'A' or a 'frame' 'B'. Again, the choice is made alternately and is enforced by a subnet. The type 'buffer' is shown below, 'frame' 'A' is distinguished by dotted arcs, and the alternating choice mechanism of 'frame' 'A' or 'B' by solid arcs.



The processes use operations on two instances of a buffer called 'tokens' and 'I-code'. In the following diagram, the individual instances of buffer are distinguished by dotted and dashed arcs respectively. The processes are distinguished by solid arcs.


lexical_analysis
input_source_text.
tokens.A.fill
tokens.B.fill
tokens.A.empty.
tokens.B.empty.
prepare_for_further_tokens.
semantic_analysis.
produce_I-code.
further semantic_analysis.
I-code.A.fill.
I-code.B.fill.
I-code.A.empty
I-code.B.empty.
output_code.
code_production.

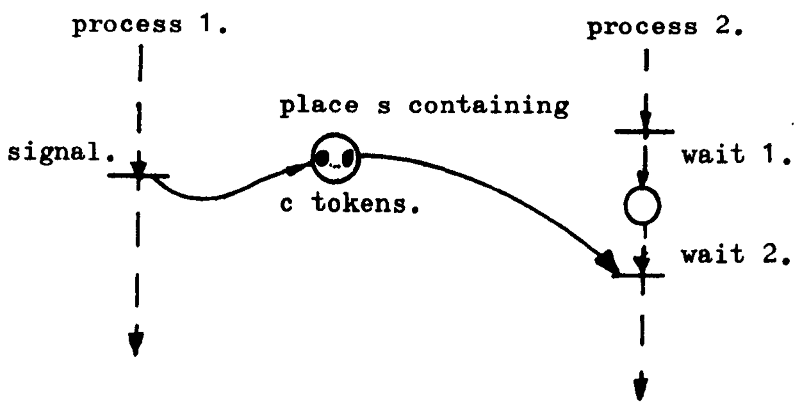
In this last example, the structure of the program into various objects corresponding to type definitions is reflected in the final simulating net. The connections from such objects to the surrounding net are uniform with respect to their original type definition as a result of the restriction that connections of such a subnet can only be made through operations to its surrounding net. Hence, properties that can be shown about the original type definition may be applied to instances of the type definition, given that the operations are used in the correct manner.

4.5 CORRECTNESS OF IMPLEMENTATION OF E-PATH EXPRESSIONS.

We can now show that the implementation described in Chapter 3 using semaphores and P, V operations is correct with respect to the presentation that we have described for paths and processes in that they both generate the same sequences of actions. We shall use the realization given by Lautenbach and Schmid of a semaphore and the operations upon it.

Petri net representation of a semaphore (Lautenbach and Schmid, 1974).

Lautenbach and Schmid describe a net representation for a semaphore which corresponds to the informal definition given by Dijkstra (Dijkstra, 1968a). From their Petri net representation, they derive the semaphore theorem of Habermann (Habermann, 1972) which describes a special invariant characterizing the way in which the primitives work. The net they give is as follows:-



Lautenbach and Schmid write:-

'In this Petri net the semaphore is represented by the place s. The standard signal (s) operation corresponds to the firing of the transitions "signal" and possibly "wait2", and the wait(s) operation to the firing of the transitions "wait1" and possibly "wait2", whereby it is presumed that the transition "wait2" fires as often as possible.'

They give the semaphore theorem as:-

Number of firings of wait2 = minimum(number of firings of wait1,
C + number of firings of signal),

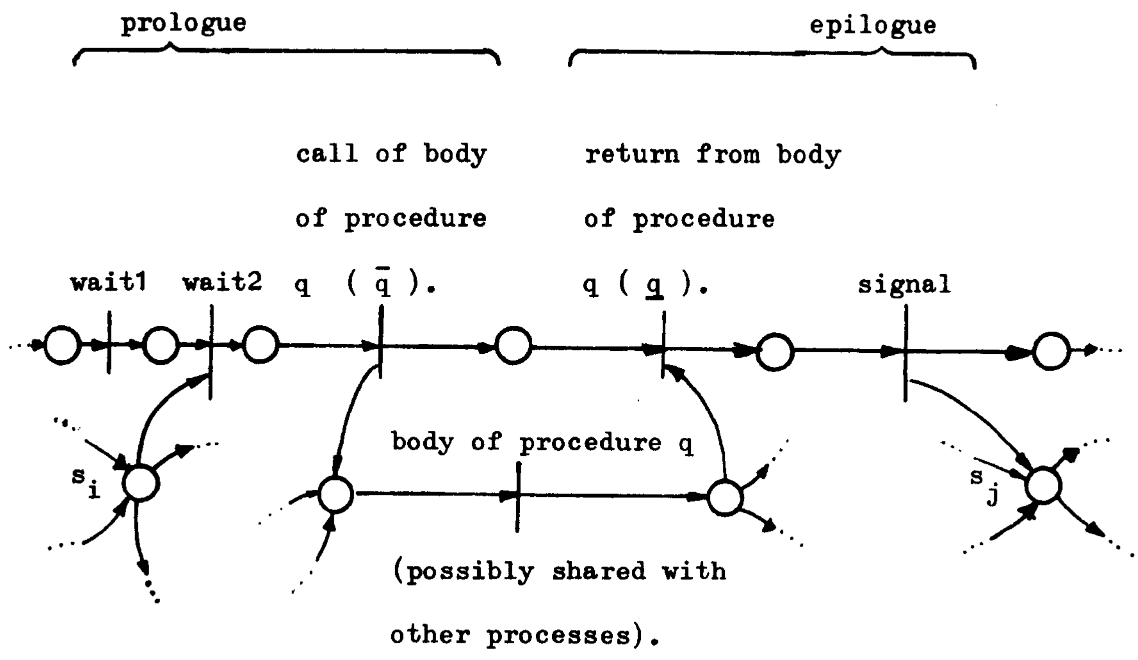
where C is a constant representing the initial value of the semaphore.

<u>Petri net representation of the P, V implementation of a procedure synchronized by a path expression.</u>

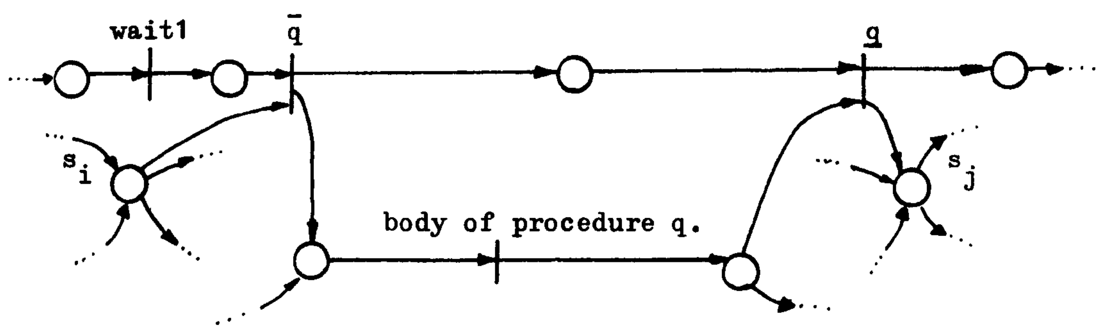
Suppose we have a procedure q synchronized by an E-path expression. The general form of that procedure, when implemented using the mechanism described in Chapter 3, is as follows:-

<u>procedure</u> q: <u>begin</u> P(s_i); (body of procedure q); V(s_j) <u>end</u>;

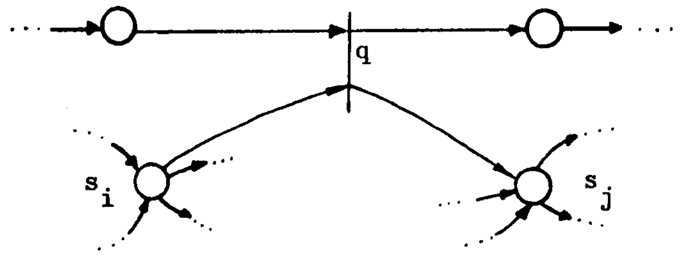
where P(s_i) is in the prologue of q and V(s_j) is in the epilogue. We shall represent the P operation as occurring at the beginning of the procedure prologue, before the procedure call mechanism. Similarly, the V operation will occur at the end of the procedure epilogue, after the procedure return mechanism. Only one semaphore in the E-path implementation may have a value 1. Since q begins by executing a P operation, the V operation cannot be performed on a semaphore with value 1. Hence, the binary semaphores behave as counting semaphores with values 0 and 1, permitting q to be represented as a Petri net using the subnets for a semaphore described above:-



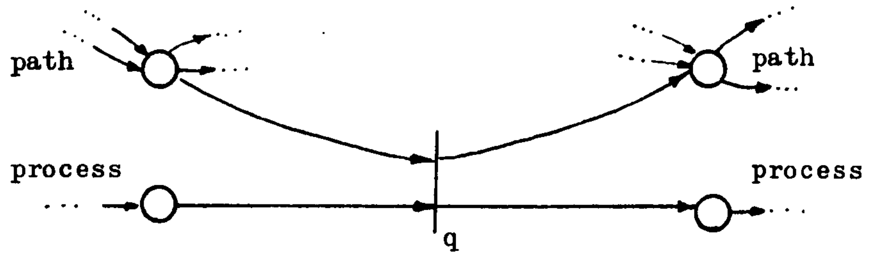
In general, the body of the procedure will be shared between many processes and the semaphores will have other P and V (wait and signal) operations performed on them by different procedure prologues and epilogues (these are represented in the diagram by additional arcs to and from s_i, s_j and the body of the procedure q). From abbreviation c) we may contract 'wait2' and the call of procedure q ($\bar{q}$) to a single transition. Similarly, the return from procedure q ($\underline{q}$) and 'signal' may be contracted to a single transition:-



Finally, we may apply the abbreviation c) to wait1 and the new $\bar{q}$ to give a transition q', and the abbreviation d) to $\bar{q}$' and $\underline{q}$ and the body of q to give the transition q in a net:-



By lemma 4.2, these abbreviations will not affect the behaviour of the surrounding net. However, TR4.3 gives for the same procedure q the following characterization:-



The characterization we have obtained from abbreviating the Petri net representation of the implementation scheme for procedure q is identical to that given by TR4.3. In addition, our earlier analogy in Chapter 3 between semaphores and states (places in the E-net of an E-path) is clearly shown by the correspondence of the places in the path above to the semaphores s_i and s_j.

4.6 SCHEDULING POWER.

Associated with a given synchronization specification is an implicit mechanism which resolves conflicts between alternate sets of sequences of actions which are permitted by that specification. A conflict is identified in the simulating net of a path expression and process by transitions which are in conflict over places. (See Chapter 2 for a definition of conflict in a Petri net). Lipton defines a scheduler for a generalized parallel process as a predicate on the timings of that process, where a timing is a finite or infinite sequence of actions of that process (Lipton,1973). The implementation of a synchronization specification (whether the specification is given by a path expression or a P or V operation on a semaphore) must permit exactly the sets of sequences of actions which are permitted by the specification and must also include a scheduler which will resolve possible conflicts in that specification. Such a scheduler will be called the implicit scheduler of the synchronization specification. Suppose we have two synchronization notations N1 and N2. The scheduling power of N1, denoted SP(N1) is at least that of SP(N2) if every expression E2 of N2 has an implementation in which all the conflicts of E2 are expressed as conflicts of specifications written in N1. That is, if the conflicts of N2 can be resolved by the implicit scheduler of N1. In addition, we define:-

SP(N1) equals SP(N2) if SP(N1) is at least SP(N2)
and SP(N2) is at least SP(N1).

SP(N1) is greater than SP(N2) if SP(N1) is at least SP(N2)
but there exists at least one conflict in an expression of N1
which cannot be expressed as conflicts in expressions of N2.

As an example of the scheduling power of the E-path notation, we shall construct a semaphore notation which has equal scheduling power. The semaphore has two states (corresponding to 0 and 1), but unlike a binary semaphore a V operation may only be completed if the semaphore is in the state corresponding to 0. This semaphore notation may be used to construct critical sections and to signal between processes which are the basic requirements outlined by Dijkstra for constructing synchronization and co-ordination between concurrent processes (Dijkstra,1968b). The semaphore is described by the type definition (Chapter 1) as:-

```
type semaphore;
    path ( V ; P )* end;
    procedure P:  null;
    procedure V:  null;
operations P, V
endtype;
```

Declarations of the above type:-

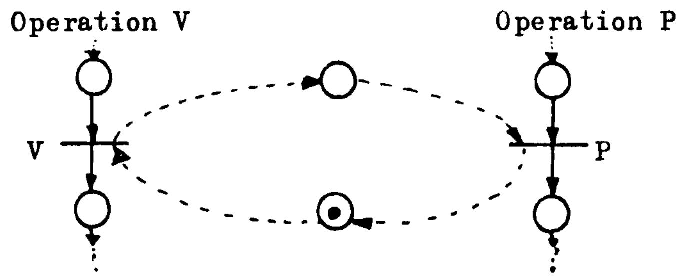
semaphore s;

introduce semaphores which have an initial 'value' zero. Semaphores with initial 'value' one may be introduced by declaring a further type semaphore$_1$ whose path expression has the positions of P and V reversed. Synchronization expressions written in this notation will have the form:-

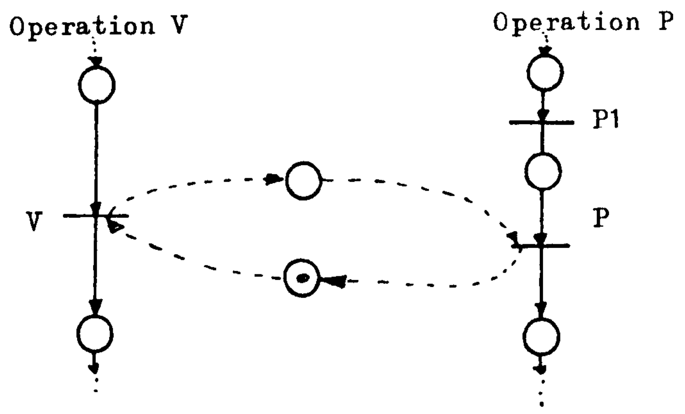
... s.V .... and ... s.P ... .

The scheduling power of the E-path notation is at least that of this semaphore notation because the implementation expresses every conflict which may arise in a P or V expression as a conflict in an E-path expression. Suppose, however, we wished to implement E-path expressions in terms of this semaphore notation. Trivially, we may substitute this semaphore notation for the binary semaphore notation we used in 3.5 to implement E-paths. Thus, we conclude that the scheduling power of this semaphore notation is at least that of the E-path notation, and hence that the two notations have equal scheduling power.

Finally, we will show that this semaphore notation conforms to the semaphore theorem of Habermann (Habermann,1972). The type semaphore above can be represented by the following simulating net using the ideas of 4.4:-



(To distinguish the simulating net of the E-path, its arcs are dashed.)
The input place to transition P is only an input place to transition P (a process cannot request a P operation on more than one semaphore at a time) and we may introduce an additional transition before P which may be removed by application of abbreviation c) without affecting the synchronization specification. This gives the net:-



The Petri net of a semaphore given by Lautenbach (4.5) is identical to the above net except for directed arcs from P to V and the additional marked place. Lautenbach derives from such a net the semaphore theorem of Habermann. The extra arcs and place in the above net impose the following additional restriction on the firing of P and V:-

$$0 \leq \text{number of Vs} - \text{number of Ps} \leq 1.$$

This restriction corresponds to the difference between this semaphore notation an and a counting semaphore notation.

4.7 A CLASSIFICATION OF PROCESSES.

A study of general processes and their interactions with path expressions would be a substantial task. However, several subclasses of processes may be identified and are useful. These subclasses of processes allow certain aspects to be studied. The classification of processes will be restricted to ones that may be described by regular expressions. The first classification we shall make is to describe E-processes (these processes resemble E-paths) and show that a criterion exists which will guarantee freedom from deadlock for a program consisting of such processes and E-paths. In later chapters we shall introduce other classes of processes which have differing properties.

E-processes.

We will characterize E-processes by means of a set of productions and restrictions using a similar technique to that used to describe E-paths.

E-process production rules.

p1:	E-process	::=	process (p-sequence)* end
p2:	p-sequence	::=	p-unit [; p-sequence]
p3:	p-unit	::=	p-element * ; p-element
		|	p-selection
p4:	p-selection	::=	p-element [, p-selection]
p5:	p-element	::=	procedurename
		|	(p-selection [; p-sequence])

Restriction R1: No procedurename may occur more than once in a process.

Restriction R2: No procedurename may occur in more than one process.

The two restrictions and the productions are almost identical to those for E-paths and serve a similar purpose. Thus, no procedure name may be executed under more than one sequencing condition. The corresponding state machine to an E-process can be constructed by treating the expression α found in process α end as a regular expression and applying the construction 2.19. Because of the similarity between E-path expressions and E-processes we may use 3.1 to simplify the construction, and the results obtained in Chapter 3 for E-paths and their corresponding state machines also apply to E-processes and their corresponding state machines. The simulating net for an E-process is trivially generated by constructing the State Machine Petri net which is equivalent to the state machine corresponding to that process.

Procedure declarations.

E-processes may execute procedures which have been given procedure bodies. A procedure body is introduced by the following declaration:-

p6: E-procedure ::= <u>procedure</u> identifier : p-element

where p-element is the body of the procedure with name 'identifier'. The productions for 'p-element' are those given in p5 above. The following restriction applies to the use of procedures with E-paths and E-processes:-

<u>Restriction</u> R3: No procedure named in a path expression may, during its execution, invoke by a series of other procedure calls, another procedure named in that path expression.

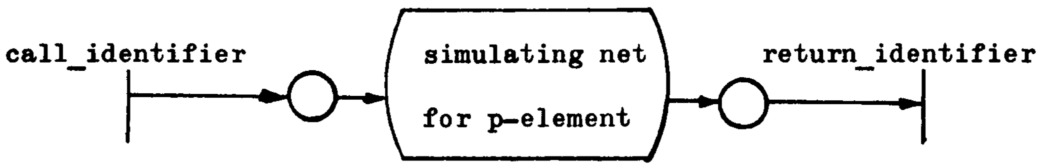
Restriction R3 is a necessary condition imposed to prevent inappropriate procedure invocations leading to a deadlock. Further discussion of this restriction and of hierarchies of procedure calls involving path expressions is postponed until later in this chapter. The restrictions R1 and R2 for E-processes include the procedure bodies that the processes may execute. Thus, no procedure may be executed under more than one sequencing condition. Alternatively, E-processes are constructed so that constituent actions need not be implemented as true procedure calls using a mechanism to represent the 'control flow', as would be necessary if identical actions could appear more than once in a given process or in several processes. Instead, they may be implemented by a simple 'macro expansion' and substitution. This allows us to reserve the procedure call representation for more complex processes, and will permit discussions of E-process behaviour to be simple. As a conclusion to these remarks, a procedure body may be directly substituted for the occurrence of the corresponding 'procedurename' in the appropriate process.

The procedure declaration introduces a regular expression 'p-element' which has a corresponding state machine and simulating net representation.

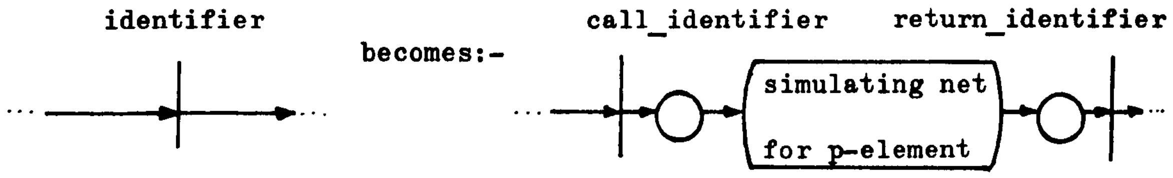
The simulating net for a procedure body is the simulating net for 'p-element' enclosed between an event representing the invocation of the procedure, and an event representing the return from the execution of that procedure. The procedure declaration:-

procedure identifier: p-element

gives a procedure body:-



which may be directly substituted in an E-process for the transition labelled with the procedure name 'identifier'. (That is,



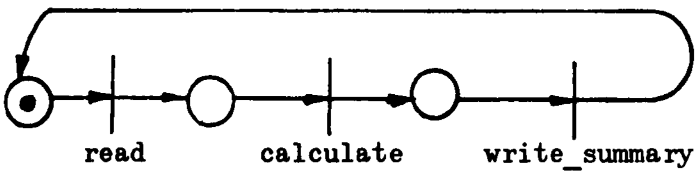
The representation of a procedure invocation by an E-process is a simplification of that given in 4.2; the place representing the condition that a procedure call has been made is omitted.)

Example of an E-process.

Suppose that we have the following process:-

process (read ; calculate ; write_summary)* end

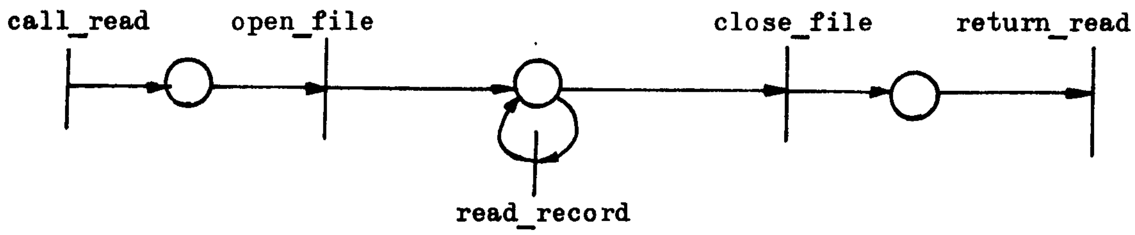
This has the simulating net:-



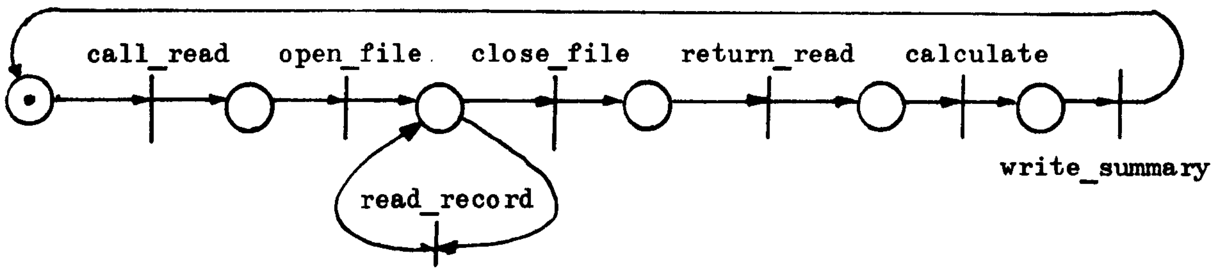
Procedure read may not be an atomic action and perhaps has a body defined by the following procedure declaration:-

<u>procedure</u> read: (open_file ; read_file ; close_file)

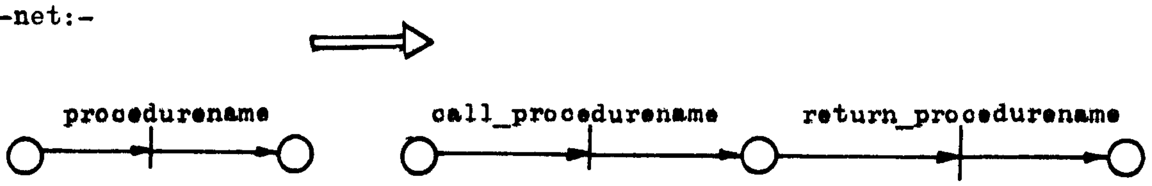
which has the simulating net:-



Substituting the procedure body of read for read in the process gives:-



Finally, for procedures whose bodies have been given by p6 and which occur in an E-path, we make the following replacement in the body of the E-net:-



<u>Example of a simulating net for a set of E-paths, E-processes and procedures.</u>

The set of E-paths, E-processes and procedures:-

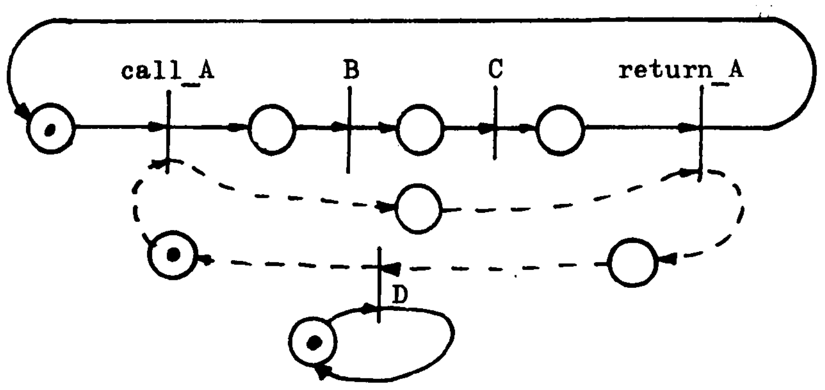
<u>process</u> (A)* <u>end</u> ;

<u>process</u> (D)* <u>end</u> ;

<u>procedure</u> A: (B ; C) ;

<u>path</u> (A ; D)* <u>end</u>;

has the simulating net:-



where the subnet corresponding to the path has dashed arcs. Note that the dashed arcs and place between call_A and return_A can be removed without affecting the synchronization of the net.

<u>Type definitions</u>.

Types (as in Chapter 1) may be used in a program to introduce paths, processes and procedures to represent objects. Instances of types have their locally defined operations, procedures and actions identified and distinguished from other instances by the use of the Simula dot notation. Types for use with E-paths and E-processes are as follows:-

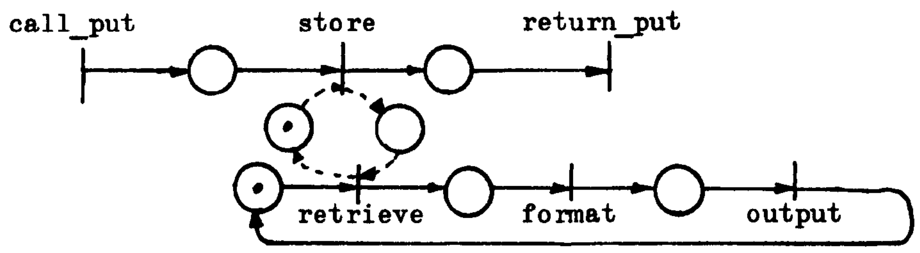
<u>Production rules for types</u>.

p7:	type	::=	<u>type</u> identifier ;
			⟨collections of types, paths, procedures, processes and instances separated by semicolons⟩ ;
			<u>operations</u> < list of procedures >
			<u>endtype</u>

The <u>simulating net for a type definition</u> is the simulating net for the constituent path expressions and processes. For example, the simulating net of the type definition:-

```
type virtual_output_device;
    path ( store ; retrieve )* end;
    process (retrieve ; format ; output)* end;
    procedure put: ( store ) ;
operations put
endtype;
```

is:-



where the dashed lines represent the simulating net of the path expression.

Instances of a type are introduced as follows:-

```
p8:  instance          ::=  ⟨name_of_type⟩ ⟨identifier list separated by
                                                  commas⟩
```

The simulating net of an instance is a copy of the simulating net of the corresponding type definition in which the labels on the transitions of the net given by the type definition are replaced by the name of the instance followed by a dot and the original label (see 4.4).

Finally, a program is given by the production:-

```
p9:     program  ::=   begin ⟨collections of types, paths, processes
                             procedures  and instances separated by
                             semicolons.⟩
                       end.
```

4.8 EXAMPLES OF PATHS AND PROCESSES.

The following set of examples illustrate programs, types, E-paths and E-processes.

1. Synchronous processes (Bekkers, 1974).

The following example was given by Bekkers and concerns the problem of synchronizing two processes so that, at one stage of their execution, each process must wait for the other process and the processes can then restart together. For this particular solution, the processes are aware of their separate identities. (In Chapter 5 we will give a solution in which this is not the case.) The synchronization is incorporated in a type definition called synchro:-

```
type synchro;
    path ( arrive1 ; depart2 )* end;
    path ( arrive2 ; depart1 )* end;
    procedure synk1: ( arrive1 ; depart1 );
    procedure synk2: ( arrive2 ; depart2 );
operations synk1, synk2
endtype;
```

If process 1 invokes synk1, it will execute arrive1 and then wait to execute depart1. It will only be able to proceed after the process 2 has executed arrive2. If process 2 executes synk2 before process 1 has executed synk1, then it must wait to execute depart2.

2. Producer/Consumer processes.

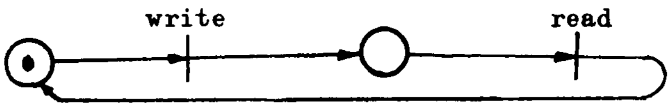
The following examples describe several buffer systems in which two processes, a producer and a consumer, communicate by placing messages in two shared message slots.

In the first example, the producer process creates a message by executing either a procedure produce or produce', and places such messages alternately in one slot and then the other. The consumer process removes a message from each slot alternately, and consumes such messages by executing either the procedure consume or consume'. Using the requirements for a buffer which were given in 1.2 with respect to its integrity, the following program may be constructed:-

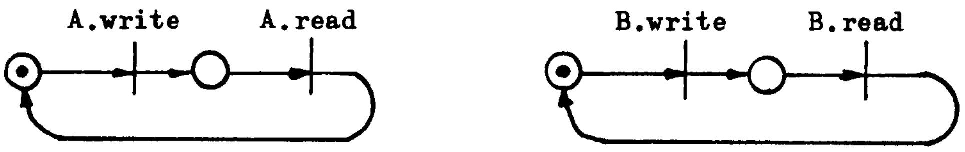
```
begin
    type slot;
        path ( write ; read )* end;
    operations read, write
    endtype;
    slot A, B;
    process ( produce ; A.write ; produce' ; B.write )* end;
    process ( A.read ; consume ; B.read ; consume' )* end;
end.
```

where read and write are procedures which perform reading and writing on the buffer, the details of which, for brevity, we have omitted.

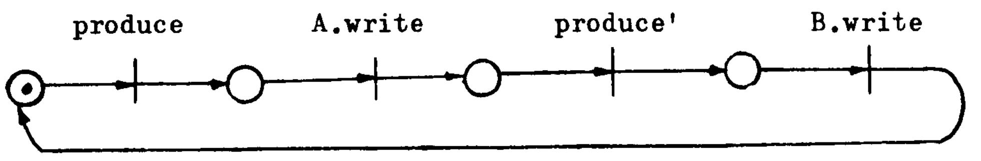
The type definition of slot has a simulating net:-

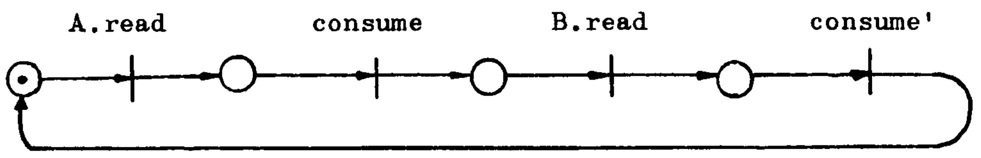


Two different instances of the type are created by the instantiation, these nets being formed from the net description of the type by prefixing the names with the identifier of the instance:-

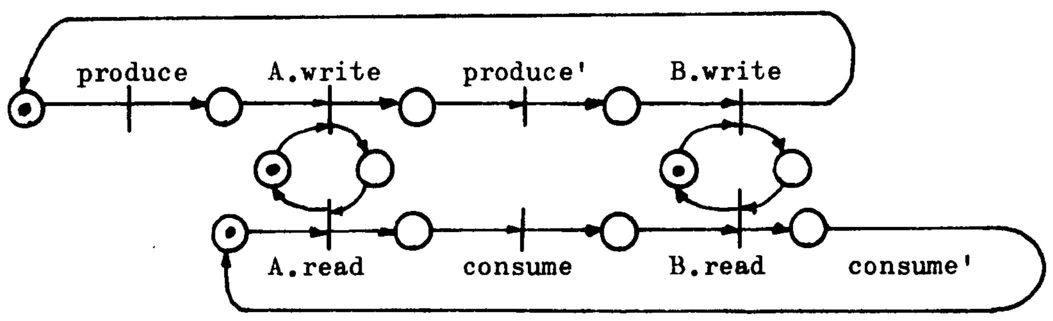


Each individual process gives rise to a net as follows:-





Finally, we take the intersection of the process nets and path nets giving:-



Considering each individual slot, since they are described by only one path expression, Theorem 4.1 implies that the read and write operations are live and safe:- that is they are mutually exclusive and it is always possible to have a sequence of process invocations which will allow any one of them to be executed. It is trivial also to verify that the final program is live and safe. The slots will always preserve the integrity of the data they contain and synchronize the procedures reading and writing correctly.

Let us now consider the buffer as a whole. It consists of two slots and is dependent on the selection of the correct slot by the program. For example, the following program deadlocks:-

```
begin
    type slot;
        path ( write ; read )* end;
    operations write, read
    endtype;
    slot A, B;
    process ( produce ; A.write )* end;
    process ( B.read ; consume )* end;
end.
```

Another example of incorrect behaviour, but one which is less noticeable, concerns the degree of parallelism permitted in a program. For example, we have assumed in having two slots in the buffer that they might be used in parallel. Suppose, however, that the buffer is used in the following way:-

```
begin
    type slot;
        path ( write ; read )* end;
    operations write, read
    endtype;
    slot A, B;
    process produce ; A.write ; produce' ; B.write end;
    process B.read ; consume ; A.read ; consume' end;
end.
```

then reading and writing can only take place in mutual exclusion.

To demonstrate the correct behaviour of the buffer in the first example, we need to know about the processes using it. To continue a theme of the thesis we observe that if the selection of the slot is left to the buffer to decide, then the buffer can be designed so as to be correct, irrespective of the use that is made of it by the program in which it is declared. Such a buffer will use the slots alternately for writing and reading.

The following type buffer will enforce the correct discipline on the use of the slots by means of two paths. We shall assume that the type slot has already been declared.

```
type buffer;
    type ring;
        slot A, B;
        path ( read1 ; read2 )* end;
        path ( write1 ; write2 )* end;
        procedure read1: ( A.read );  procedure read2: ( B.read );
        procedure write1: ( A.write ); procedure write2: ( B.write );
    operations read1, read2, write1, write2
    endtype;
    ring R;
    procedure write': ( R.write1 , R.write2 );
    procedure read':  ( R.read1 , R.read2 );
operations read', write'
endtype;
```

The type buffer has two operations read' and write' and these permit the buffer to be used by processes as if it contained a single slot. The net representing each slot is as before, and the net representing the type buffer can easily be constructed. The type mechanism provides a structuring mechanism both for the program and for the simulating net for that program and hence is useful in designing and studying the behaviour of programs. This last example is similar to the ringbuffer example of Chapter 1 for a ring with two slots. The selection of a slot in procedures write' and read' and the path expression synchronizing the use of the slots correspond to the pointer mechanism of the earlier example.

The examples we have presented are simple and are easy to analyze. For more complex programs we could use the formal proof techniques of their properties using the mathematics and research of, for example, Petri (Petri 1962), Lautenbach (Lautenbach, 1973), and Holt and Commoner (Holt and Commoner, 1970). However, such detailed investigations would be lengthy and divert attention from the main theme of this thesis.

4.9 PROPERTIES OF E-PROCESSES.

The class of E-processes has several properties which are useful in investigating program behaviour. Because of the similarity between transformation rules and restrictions of E-paths and E-processes we can assert that:-

Lemma 4.4: An E-process has a live, safe, strongly connected simulating net.

Proof: The proof follows directly by comparing the transformations, productions and restrictions for E-paths and E-processes.

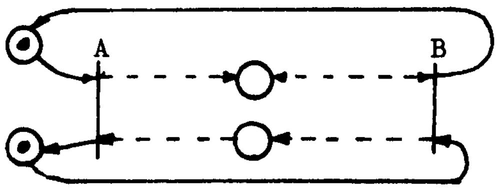
Since the restrictions we have imposed on processes do not allow procedure names to occur in more than one process, the only interactions between processes in a program of E-processes and E-paths must happen as a result of performing the intersection transformation. This transformation combines paths and processes by means of combining their transitions. Deadlock can only occur because two or more transitions have tokens on their input places but are not enabled. (This follows because each component subnet (from processes or path expression) is live, safe and strongly connected.)

Example 1.

path (A ; B)* end;

path (B ; A)* end;

This has the simulating net :-



which is dead (that is, not live). (The dashed arcs denote an empty cycle, i.e. a cycle containing no marker (2.12), and characterize the deadlock situation.)

Example 2.

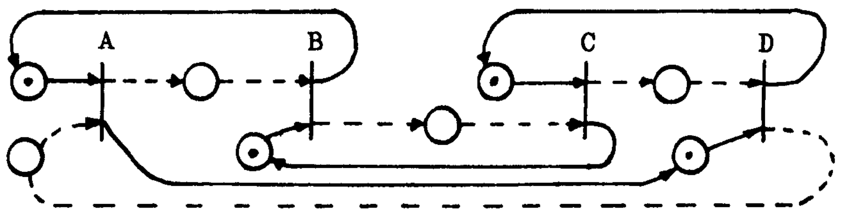
path (A ; B)* end;

path (C ; D)* end;

process (B ; C)* end;

process (D ; A)* end;

The corresponding simulating net to these paths and processes is:-



which is again dead. (The dashed arcs denote an empty cycle.)

However, there is at least one class of programs consisting of E-paths and E-processes for which we may guarantee liveness (that is, the absence of deadlock). Let Eo-paths and Eo-processes be E-paths and E-processes which contain no selections or repetitions (except for path (...)* end). The simulating net of Eo-processes and Eo-paths may be classified as follows after the application of the intersection transformation:-

Theorem 4.5: The simulating net of a program consisting of Eo-paths and Eo-processes is a safe Marked Graph (See 2.9).

Proof: First we show that such a net is a Marked Graph. Since there are no selections or repetitions allowed in Eo-paths and Eo-processes, the simulating nets will have places with only one input arc and one output arc (see 2.19 and 3.1). The intersection transformation applies only to transitions, leaving the places unchanged. Procedures are expanded by substitution and hence there are no shared procedures and no requirement for the net representation of a procedure call. Thus the procedure mechanism introduces places with only one input arc and one output arc. Therefore, the simulating net of a program of Eo-paths and Eo-processes will have only places with one input and output arc and will be a Marked Graph (2.9).

The simulating nets of individual Eo-paths and Eo-processes are safe (4.1 and 4.4). Since the sharing of procedures between processes is excluded by R2, the procedure mechanism does not affect the safeness of the simulating nets of the processes. Finally, safeness is a property of places and since the intersection transformation operates only on transitions it remains unaffected by 4.3.

Q.E.D.

The above classification allows us to choose a result concerning liveness from Petri net Theory (2.1,2.2) upon which a criterion can be based for such programs to guarantee freedom from deadlock. We must restrict such programs so that they are represented by Marked Graph simulating nets which do not contain empty cycles. Consider the following criterion:-

Criterion C4:

Given a set of Eo-paths and Eo-processes in a program, then this set cannot contain paths and processes in which there are procedures X1 to Xn forming sequences:-

```
...    X1    ...        ;        ...    X2    ...
...    X2    ...        ;        ...    X3    ...
        .                                .
        .                                .
        .                                .
...    Xn    ...        ;        ...    X1    ...
```

Example. The following program does not meet the criterion:-

```
path ( A ; B )* end;
process ( B ; C ; D ; E )* end;
path ( C ; G ; H )* end;
process ( H ; A )* end;
```

since it contains the sequences:-

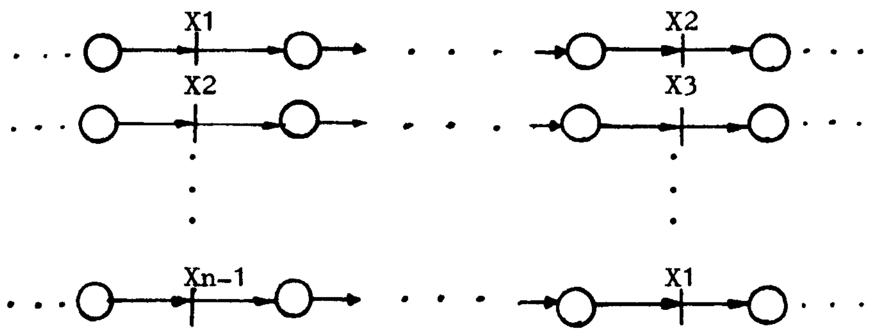
```
A        ;        B
B        ;        C
C        ;        H
H        ;        A
```

Examples 1 and 2 above also do not satisfy the criterion.

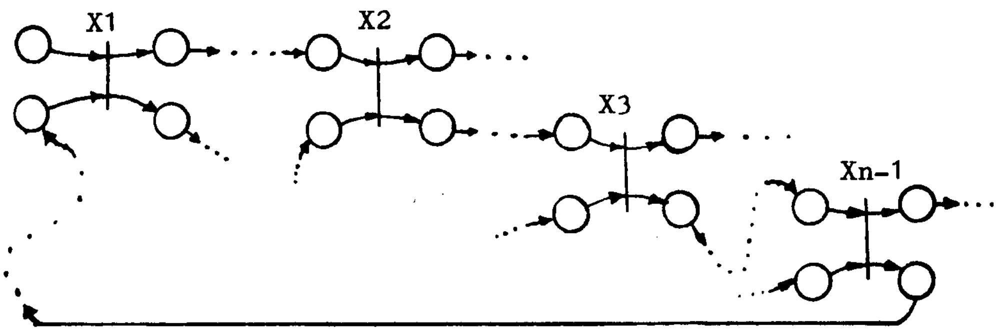
Theorem 4.6. The criterion C4 is a necessary and sufficient condition for programs consisting of Eo-paths and Eo-processes to have a live and safe Marked Graph simulating net.

Proof: The simulating net of a program which meets C4 is a safe Marked Graph Petri net from Theorem 4.5. A necessary and sufficient condition for a Marked Graph to be live is that it contains no empty cycle (2.12). The individual simulating nets of the Eo-paths and Eo-processes are live by 4.1 and 4.4. The intersection transformation applies only to transitions, and leaves the places unchanged. Because there are no shared procedures in E-processes, there is no requirement for a net representation of a procedure call and no further interconnections between processes. Hence, empty cycles can only be generated by the intersection transformation. It is trivial to verify that a program of Eo-paths and Eo-processes containing a forbidden set of sequences under C4 will have a simulating net which has an empty cycle:-

The set of paths and processes will have individual simulating nets as follows:-



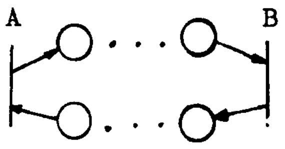
which become, under the intersection transformation:-



(The dotted lines between places indicate that the places belong to the same component simulating net.)

Thus, the necessary condition is established.

Suppose an empty cycle exists in the simulating net of a program of Eo-processes and Eo-paths. The circuit must have been formed by the intersection transformation since all circuits in the component nets have places which are live (4.1, 4.4). Suppose we have an empty cycle:-



Then there must be a set of sequences from A to B, and a set of sequences from B to A and we conclude that C4 is not met.

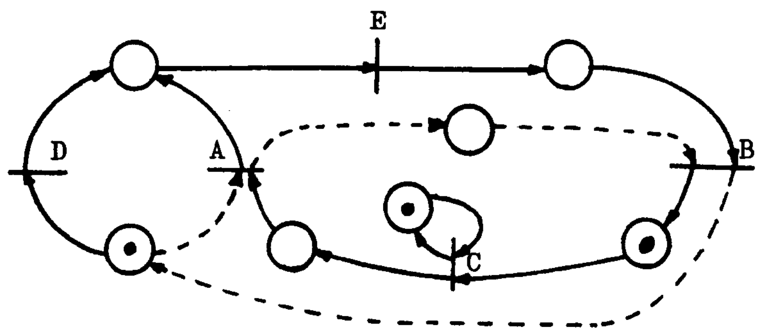
Thus, the sufficient condition is established.

Q.E.D.

In general, however, more complicated algorithms must be used to detect deadlock in E-paths and E-processes. For example:-

```
path ( C ; A ; B )* end;
process ( (A , D) ; E ; B )* end;
process ( C )* end;
```

deadlocks. The simulating net is as follows:-

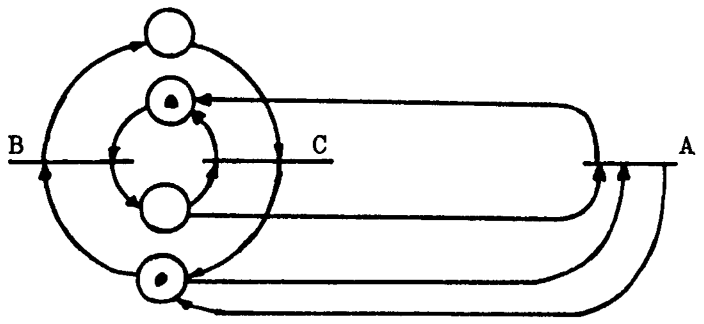


Here, although no empty cycle is initially present, one may form if a process executes the procedure D. The empty cycle which then results is indicated in the diagram above by dashed arcs.

The next example has a dead transition A which corresponds to starvation (Dijkstra,1968a):-

```
path ( (B ; C) , A )* end;
process ( B ; ( C , A ) )* end;
```

with simulating net:-

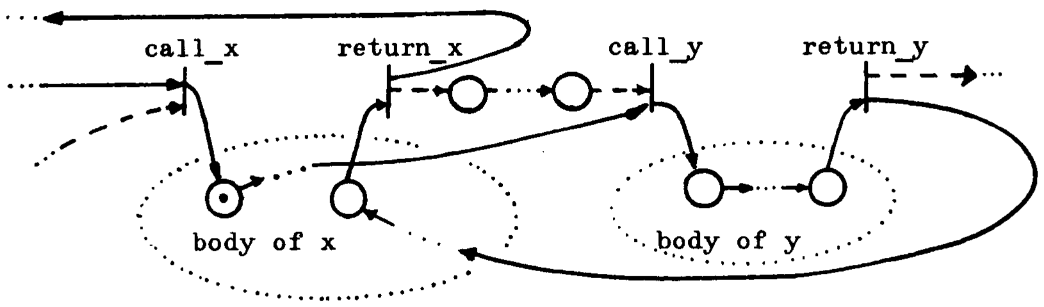


4.10 HIERARCHIES OF PROCEDURE CALLS INVOLVING PATH EXPRESSIONS.

So far in this chapter, we have discussed interactions between paths and processes without investigating the effects of procedures within path expressions invoking other procedures also named within path expressions. First, we shall show that restriction R3 (no procedure named in a path expression may, during its execution, invoke by a series of other procedure calls another procedure named in that path expression) is a necessary condition to prevent deadlock.

Theorem 4.7: Restriction R3 is a necessary condition to prevent deadlock.

Proof: Let X and Y be procedures named in the same path expression such that X, during its execution, invokes (perhaps by a series of procedure invocations) the procedure Y. Suppose a process has invoked X and is about to execute Y. We have the following marking on the simulating net of the path and process:-



The simulating net corresponding to the path expression is marked with dashed lines, and that of the process is marked with solid lines. Call_x has been enabled by both process and path and has fired. Eventually, the process puts a marker in the input place of call_y. The other input place of call_y must receive the token generated by the firing of return_x. However, return_x cannot fire until call_y and return_y have fired. Hence, we conclude that call_y cannot fire, and the process is deadlocked.

Q.E.D.

Sometimes it will be impossible to write a set of synchronization and co-ordination constraints in one path expression (see the Readers and Writers problem in (Campbell and Habermann, 1974)). Further, it may be of advantage structurally and in design if synchronization constraints can be written separately but applied to one particular set of operations.

Example.

A file system has four operations:- read, write, open and close. The file system requires that these operations are synchronized for use in a parallel processing environment in the following way:-

path (open ; (read , write) ; close)* **end**

For one particular use of the filing system as a buffer we require:-

path (write ; read)* **end**

However, an E-path expression of this kind would contravene R2, the restriction on repeating a name in more than one path. In this example, we may impose a hierarchy of procedure calls so that both synchronization constraints must be obeyed by defining two new operations read' and write' and a path expression:-

path (write' ; read')* **end**

where

procedure write': write;

procedure read': read;

The filing system, used as a buffer, now has operations open, read', write', and close. However, the filing system may still be used separately with read, write, open, and close.

In general, when a set of path expressions is used in this way, care must be taken to avoid deadlock. In particular, we make the following remark:-

Let P1 synchronize executions of procedures p1 and p2.

Let Q1 synchronize executions of procedures q1 and q2.

Further, suppose p1 and q1 have procedure bodies:-

<u>procedure</u> p1: (body invokes q2)

<u>procedure</u> q1: (body invokes p2)

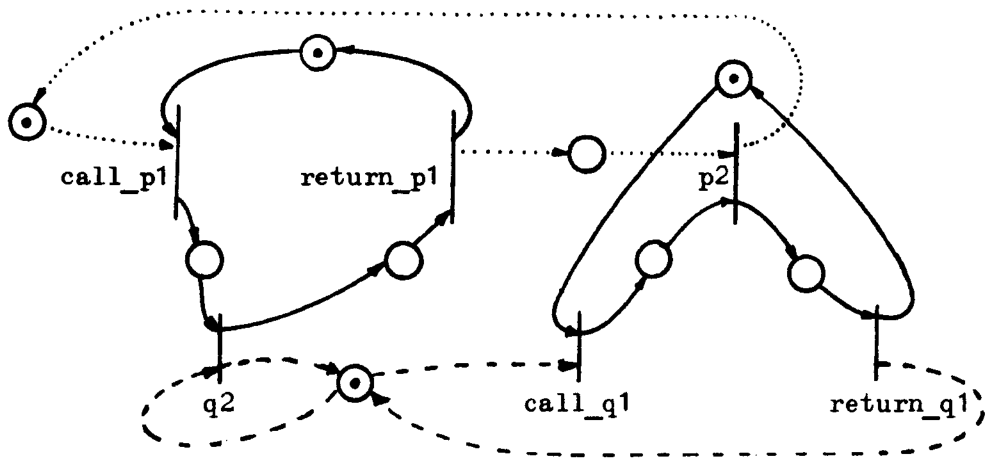
Then, if p1 and q1 are executed simultaneously by two separate processes, a deadlock may occur.

<u>Proof</u>: Suppose, the process executing p1 invokes q2. If a second process is executing q1, then the path expression Q1 prevents the first process from immediately executing q2. The first process is suspended awaiting completion by the second process of the procedure q1. Now suppose that the second process executing q1 invokes p2. The path expression prevents this process from immediately executing p2 and it is suspended awaiting completion by the first process of the procedure p1. Hence, the processes are deadlocked.

The above remark is illustrated by the following example:-

```
path ( p1 ; p2 )* end;
path ( q1 , q2 )* end;
procedure p1: q2;
procedure q1: p2;
process (p1)* end;
process (q1)* end;
```

The simulating net of this set of paths and processes is shown below:-



where the simulating nets of the processes have solid arcs, the net of the path P1 has dotted arcs, and that of the path Q1 has dashed arcs. If both call_p1 and call_q1 fire, then deadlock ensues. Return_p1 and return_q1 may not fire since, although both will have one input place which is marked, they will have one input place which is empty. Similarly for p2 and q2. Since there are only four tokens in the net, this results in a deadlock. However, the deadlock does not necessarily happen since the execution of procedure p1 may always precede that of procedure q1.

4.11 SUMMARY OF PROCESSES AND PATH EXPRESSIONS.

In this chapter we have described a representation for processes which allowed us to investigate the relation between paths and processes. We were able to show that the E-path implementation of Chapter 3 is correct with respect to this representation. A representation for types was introduced, and this illustrated the structuring properties thatthe types afford the description of objects, and in particular, when used with path expressions, of shared objects.

An interesting class of processes was described which has useful properties with respect to investigating program behaviour. Indeed, for a subset of the class of paths and processes considered, a criterion exists which guarantees the absence of deadlock. Finally, we have outlined a few of the problems that may occur within hierarchies of procedure calls involving path expressions.

CHAPTER 5

REGULAR PATH EXPRESSIONS.

The E-paths of Chapter 3 permitted the separation of synchronization from the description of procedures and processes and could be used with types to describe shared data objects.

The remainder of the thesis investigates possible alternatives and improvements to the E-path notation. Even a simple modification to the E-path notation may encounter difficulties as we shall show in Chapter 7 where we describe a parallel synchronization construction which can be added to the notation. For example, adding the simultaneous construction of Chapter 1 to a notation may result in the notation being more difficult to understand, and more difficult to implement (see also Chapter 7).

An E-path uses a limited regular expression to specify synchronization. The limitation consists of restricting the use of the Kleene star and forbidding the repetition of a procedure name within a path. The resulting set of expressions is less powerful in terms of declaring synchronization than would be a path notation allowing the full generality of regular expressions. In this Chapter, we shall investigate such a Regular Path (R-path) notation.

5.1 REGULAR PATH EXPRESSION DEFINITION.

The R-path specifies the synchronization between the executions by processes of a set of procedures using a regular expression embedded in a path end bracket. R-paths are given by the following production and restriction:-

Production rule for R-paths.

p1: R-path ::= path α end

where α is a regular expression of procedure names denoting the order in which they may be executed by processes.

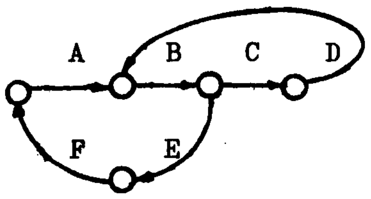
Restriction R1: A procedure name may not occur in more than one R-path.

Within an R-path, a given procedure name may be repeated several times. We shall assume that different procedures always have different names, and that therefore a repeated name in a path expression means that a given action (represented by the execution of the procedure by a process) may occur in more than one circumstance. The simulating net for an R-path will represent the multiple occurrences of a procedure name in the appropriate manner.

Examples of R-paths.

In 3.6 we remarked that non-nested selections and non-nested repetitions could not be represented using E-paths. These may now, however, be represented as R-paths:-

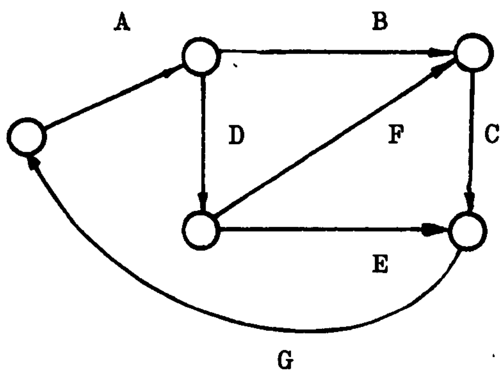
For the non-nested repetition or loop given by the state machine:-



we may write the R-path:-

path (A ; B ; (C ; D ; B)* ; E ; F)* end.

For the non-nested selection given by the state machine:-



we may write the R-path:-

<u>path</u> (A ; (B ; C) , (D ; (F ; C) , E) ; G)* <u>end</u>.

In both examples, the non-nested construction is reflected in the repetition of procedure names. Other examples of R-paths follow :-

<u>A file:</u> The following R-path describes the synchronization of the operations of a file:-

<u>path</u> (open ; (read* , write) ; close)* <u>end</u>

Following execution of 'open' by a process, either zero or more reads occur, or a single 'write'. Between opening and closing a file, a mixture of reads and writes is not permitted.

<u>A buffer</u>: The following buffer allows precisely two 'reads' to be performed for every 'write':-

<u>path</u> (write ; read ; read)* <u>end</u>

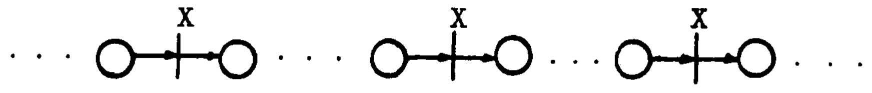
<u>R-paths as finite state acceptor mechanisms</u>.

Given an R-path expression, we may construct a finite (deterministic) state machine for the regular expression which it contains using the construction 2.19. In addition, we may minimize such state machines using the standard minimization algorithm involving the subset construction of Myhill (Myhill, 1957). Thus, we may use a finite state machine which corresponds to the regular expression of the R-path as an equivalent definition of the synchronization described by the R-path. This will be of use in the representation of R-paths as simulating nets and in the establishing of a correct implementation for R-paths in terms of other synchronization primitives.

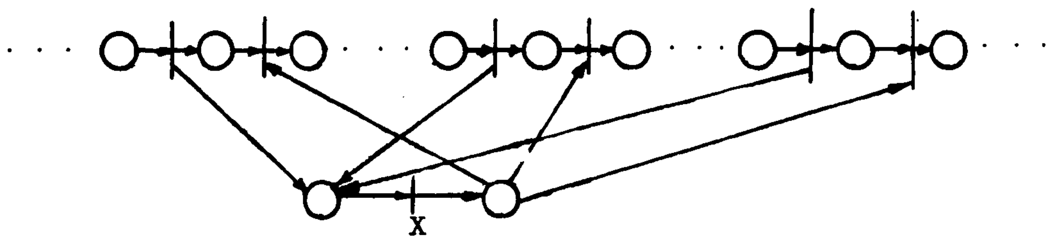
5.2 SIMULATING NETS OF R-PATHS.

As in Chapter 4, we shall describe the relationship between R-paths and processes in our model of a program by means of Petri nets. The simulating net (R-net) of an R-path expression is trivially obtained by constructing the State Machine Petri net (2.7) which is equivalent to the deterministic finite state machine corresponding to the regular expression of the R-path. However, repeated procedure names in the R-path may appear as separate transitions bearing the same label in the simulating net. These transitions will be regarded as abbreviations for a calling mechanism to a single transition to represent that particular procedure. The calling mechanism ensures the correct flow of control for the path expression in much the same way as the procedure call mechanism does for the processes in Chapter 4. The abbreviation is as follows:-

Suppose that procedure X occurs three times in an R-path, and hence the R-net has three transitions labelled X:-



These are abbreviations for the following subnet:-



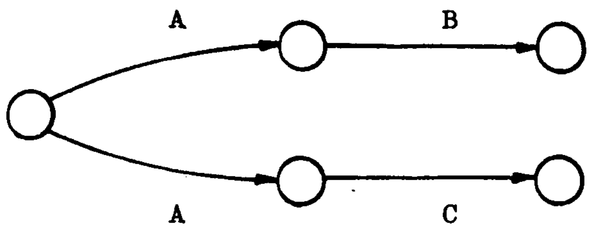
Thus, whenever X can occur in the abbreviated net, it occurs in the unabbreviated net. Although the abbreviated simulating net is a State Machine Petri net, the unabbreviated one will not be since in general such a net will have transitions with more than one output or input place (2.7). (These extra places will be introduced by the calling mechanism.)

<u>Lemma 5.1</u>: R-nets are Simple Petri nets.

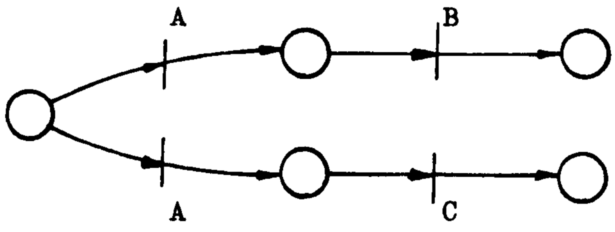
<u>Proof</u>: The calling mechanism which replaces a repeated transition name introduces two transitions to manage the call. The first of these two transitions shares one output place with all the other similar transitions from other calling points in the R-net. Similarly, the second of these two transitions shares one input place. No transition is introduced which shares more than one place. Hence, by definition 2.10, an R-net is a Simple Petri net.

Q.E.D.

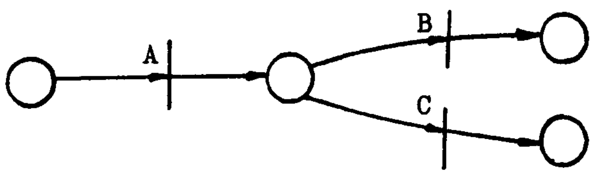
The simulating net for an R-path is constructed from the deterministic state machine corresponding to the R-path because R-paths are acceptor mechanisms for strings of procedure executions, whereas their simulating nets generate those strings of procedure executions. Consider the following state machine which is non-deterministic:-



The state machine decides whether an A should be followed by a B or a C when a B or a C is detected in the string of procedure executions being accepted. However, the equivalent State Machine Petri net:-



generates either the sequence AB or AC, and this is determined when the event A occurs. The simulating nets of R-paths will be used to determine the behaviour of paths and processes, and also to prove the correctness of an implementation for R-paths. If the Petri net corresponding to a non-deterministic state machine representation of an R-path is used as the simulating net for that R-path, then the net will make decisions before the information needed for that decision is available. For example, suppose the above net is a simulating net for an R-path and a process invokes the procedure A. In general, it will not be possible to decide whether a process is going to invoke a B or a C next. Thus, the appropriate simulating net for an R-path expression is based upon a deterministic state machine representation of that R-path. The simulating net for the synchronization given in the example above would be:-



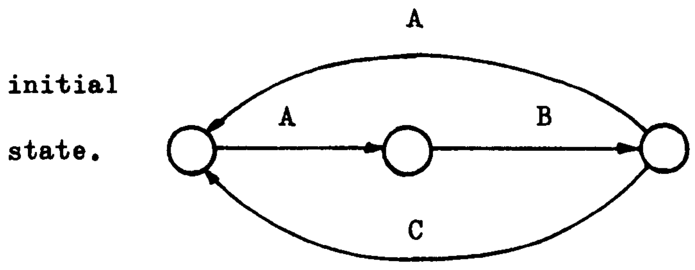
Finally, we note that, in the interests of economy, the simulating net for a given R-path should be constructed from a minimum deterministic state machine. However, whether or not they are will not in general affect the results which follow in this chapter.

Examples of simulating nets of R-paths.

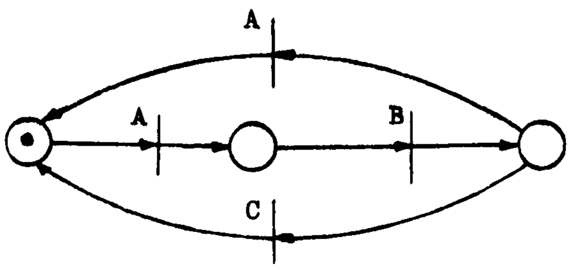
The following R-path:-

<u>path</u> (A ; B ; (A , C))* <u>end</u>

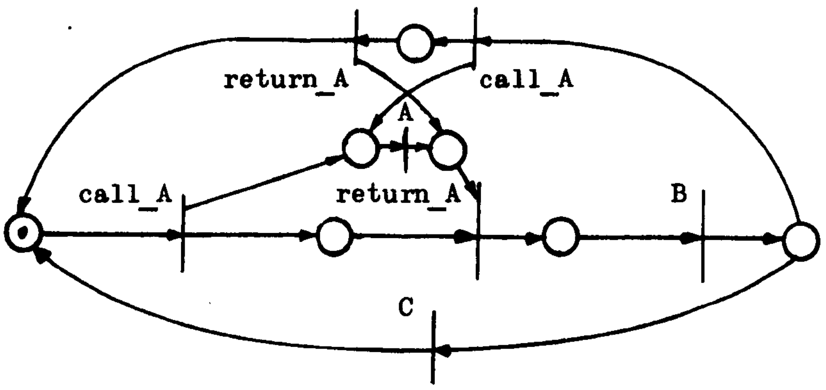
has the minimum deterministic state machine:-



The corresponding simulating net contains abbreviations for the calling mechanism to the procedure A:-



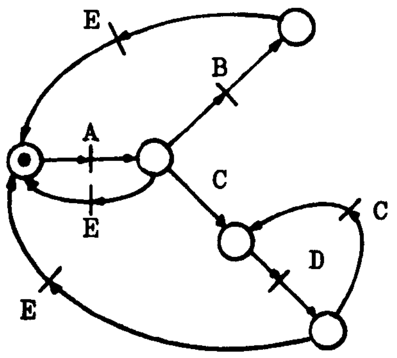
Expanding the abbreviation for the calling mechanism of the procedure A gives:-



This next example illustrates that the repetition construction may result in an abbreviated simulating net containing abbreviations for the calling mechanism. The R-path:-

<u>path</u> (A ; (B , (C ; D)* ; E))* <u>end</u>;

has the abbreviated simulating net:-



(Note that the simulating net corresponds to a minimized state machine and hence is unique up to isomorphism (2.20) with other minimized state machines with the same behaviour. Thus there can be no equivalent simulating net with the same behaviour which does not include the use of the calling mechanism. This demonstrates that the E-path notation must restrict the use of the Kleene star if the implementation described in Chapter 3 is to be used for E-path expressions.)

<u>A restriction on R-path expressions which has a useful property</u>.

Consider the following modification to the production rule for R-paths:-

p1' p1 ::= <u>path</u> (α)* <u>end</u>

where α is any regular expression. All R-paths which are written using production p1' will be restricted to describing synchronization by means of a regular expression consisting of a Kleene star embedding a further regular expression. This has the following property:-

Suppose that an R-path has accepted a sequence of procedure executions. Then there will be a sequence of procedure executions permitted by the R-path which will allow a process to execute any procedure in that R-path that it may invoke.

Thus, we have the following:-

Theorem 5.2: The transitions labelled with procedure names in the unabbreviated simulating net (Simple Petri net) of the R-path given by p1' will be live.

Proof: The abbreviated simulating net (State Machine Petri net) of the R-path is equivalent to the regular expression of the R-path with respect to its behaviour. The firing of a given transition in the abbreviated net implies, by construction, the eventual firing of the transition with the same label in the unabbreviated net. If an R-path has accepted a sequence of procedure executions, then there will be a corresponding sequence of labelled transitions which will have fired in the abbreviated net and in the unabbreviated net. Since there will be a sequence of procedure executions permitted by the R-path which will allow a process to execute any procedure in that R-path that it may invoke, we deduce that there must be a sequence of firings of transitions which will lead to the firing of a transition labelled with that name in both the abbreviated and the unabbreviated nets. However, there is only one transition labelled with that name in the unabbreviated net, and hence it must be this one that fires. Therefore, we conclude by definition 2.2 that the labelled transitions of the unabbreviated simulating net must be live.

Q.E.D.

The liveness property of the transitions labelled with procedure names in the unabbreviated simulating net of an R-path is useful in proving properties about systems of R-paths and processes. For example, if a system involving only cyclic processes and R-paths has a simulating net in which some of these labelled transitions are not live, then we may assume that the specification of the system is incorrect since certain events cannot recur. If the R-paths used in this system are not of the form given by p1', then they will introduce labelled transitions which may not be live under any combination of paths and processes. For example, consider the following system composed of an R-path and an E-process:-

<u>path</u> (A ; B)* ; B <u>end</u>

<u>process</u> (A ; B)* <u>end</u>

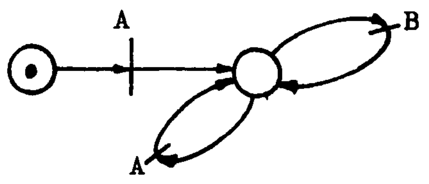
The transition labelled A in the simulating net for the system may fire and it will always be followed by the transition B firing, and vice versa. However, the following R-path gives rise to simulating nets in which the labelled transitions A and B will never be live, no matter what other processes are included in the system:-

<u>path</u> A ; B <u>end</u>

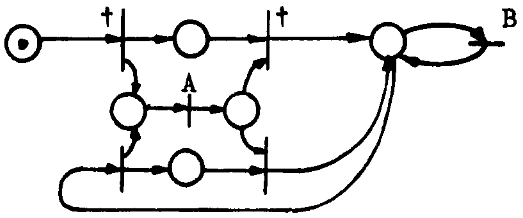
Finally, we note that not all the transitions in a simulating net of an R-path given by p1' are live. For example, the following R-path:-

<u>path</u> (A ; B*)* <u>end</u>

has the abbreviated simulating net:-



and the unabbreviated simulating net:-



which has two unlabelled transitions indicated by † which can fire only once. Note, however, that the labelled transitions are always live, as proved in Theorem 5.2.

Theorem 5.3: The simulating net of an R-path is safe.

Proof: The abbreviated simulating net of an R-path is safe by construction. The unabbreviated net is identical to the abbreviated one except for the calling mechanism. The calling mechanism increases the number of tokens on the net to two, but these tokens are on different places. The number of tokens is reduced back to one after the transition with the appropriate procedure label has fired. Hence, the net is safe. Q.E.D.

Thus, R-path expressions used in a restricted form share common programming properties with E-path expressions. The liveness of an R-net produced by p1' implies that it is impossible to write an R-path in this form which inherently contains a deadlock. Similarly, for all R-paths, the safeness indicates that it is impossible for a given synchronized procedure to be executed simultaneously by two or more separate processes.

5.3 A MODEL FOR A PROGRAM CONSISTING OF PROCESSES AND R-PATHS.

We can use a similar characterization for the combination of R-paths and processes to that used in Chapter 4 for E-paths and processes.

An intersection transformation can be used directly to give the simulating net for programs of R-paths and processes. However, R-paths allow procedure names to be repeated. Suppose that the R-net for some R-path has, when abbreviated, n transitions which correspond to n occurrences of the same procedure. Further, suppose that the process simulating nets may be abbreviated using 4.2d) so that they have r transitions representing r invocations of that procedure. Each of the r invocations of that procedure may be associated with one of the n invocations of that procedure name in the path. Hence, there will be n $\times$ r possibilities of the procedure occurring, which will be represented using the abbreviation 4.2d) as n $\times$ r transitions in the simulating net of the program. The intersection transformation is as follows:-

TR5.1 <u>Intersection transformation for the nets of processes and R-paths</u>.

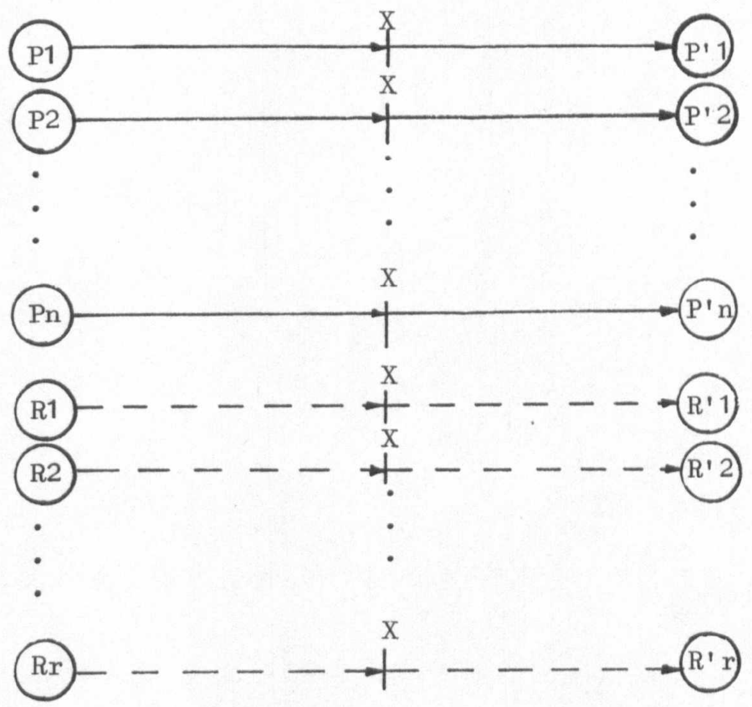
Abbreviate the simulating nets of the R-paths. A given action or procedure name will occur in only one of the simulating nets of the R-paths because of restriction R1. For each different name in the R-nets of the R-paths do the following:-

1. Identify the set of transitions in the abbreviated R-nets labelled with that name.

2. If, in the process nets, the name corresponds to the body of a procedure embedded in the procedure calling mechanism, then abbreviate the calls upon that procedure by replacing them by transitions labelled with that name using abbreviation 4.2d). Identify the set of transitions labelled with that name.

3. Apply the following transformation to the sets of transitions identified.

Replace the subnet:-

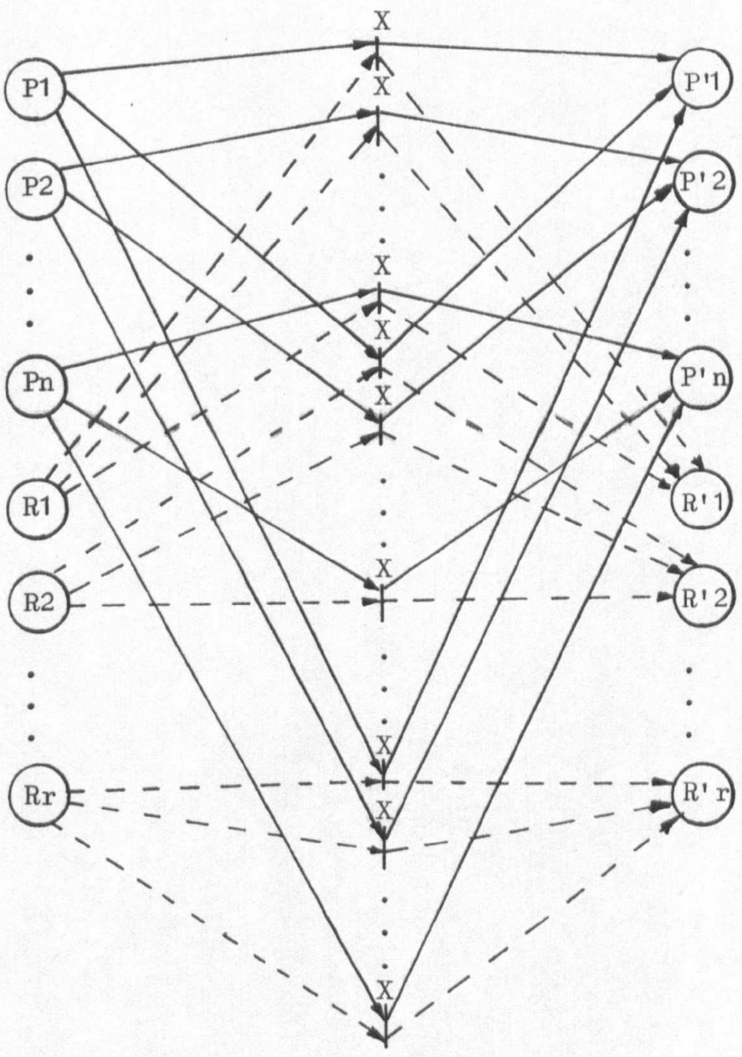


n transitions belonging to Petri net model of the processes.

r transitions belonging to a simulating net of an R-path.

(We will distinguish subnets belonging to R-nets by giving them dashed arcs .)

by:-



n × r transitions belonging to the resulting Petri net model of the program consisting of the R-path and processes.

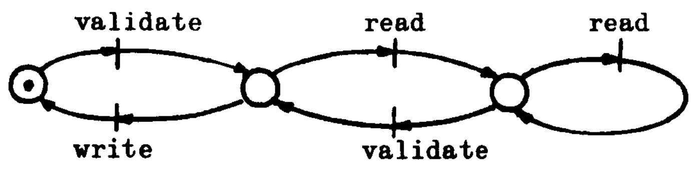
The places of the subnets involved (for clarity the places are labelled in the diagrams above) are not affected by the transformation although they may be given more output or input arcs to transitions labelled X. Each process of the n processes is connected to each of the r paths through arcs to and from the n $\times$ r transitions. The n $\times$ r transitions are all labelled with the same name and their firing corresponds to the execution of the procedure X. Thus, the transitions we have introduced above are abbreviations (4.2 d)) for the calling mechanism to a unique transition labelled X.

Example of the intersection transformation between an R-net and a Petri net representing two processes.

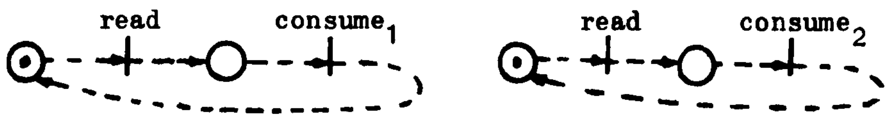
The following R-path describes operations on a file. A validate operation may be performed at any time in between reads and writes, but must be done after a write. Many reads may occur without requiring a validate operation on the file.

<u>path</u> (validate ; (read* , write))* <u>end</u>

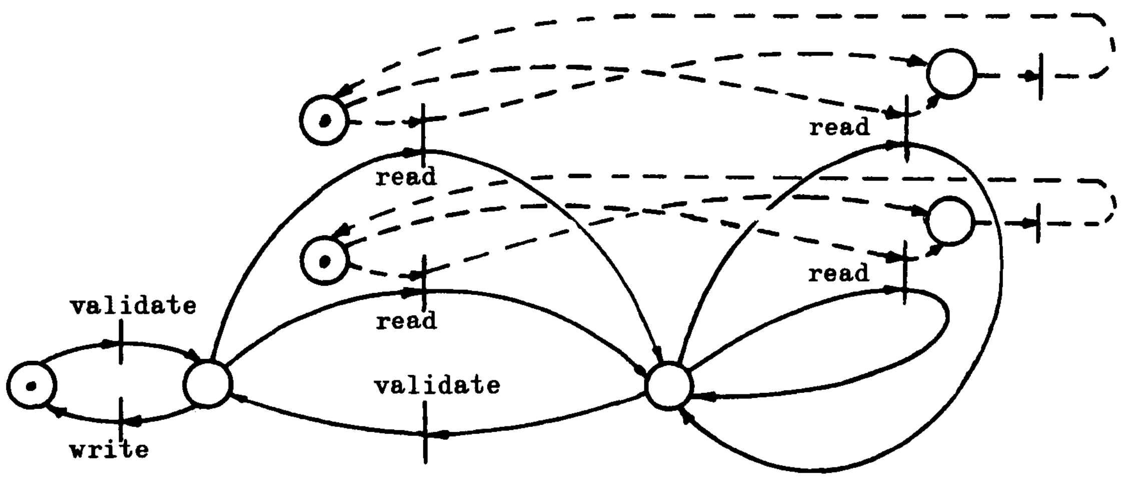
The abbreviated simulating net of this R-path is as follows:-



For the purpose of this example, we shall ignore processes which validate and write. We shall assume that there are two processes which read from the file and a model of these two processes is shown below:-



Combining the nets of the processes and the simulating net of the R-path using the intersection transformation gives:-



5.4 CLASSIFICATION OF PROCESSES.

As in Chapter 4 where we introduced an E-process as a counterpart to an E-path we now introduce an R-process as a counterpart to an R-path. R-processes are characterized by the following production and restriction:-

R-process production rule.

P2: R-process ::= process α end

where α is a regular expression.

Restriction R2: No procedure name may occur in more than one process (including the procedures that the process may invoke).

The state machine corresponding to the regular expression in the R-path can be constructed using 2.19. The simulating net for an R-process is trivially generated by constructing the State Machine Petri net which is equivalent to the state machine describing that process. This State Machine Petri net is the abbreviated net for that process, and the abbreviations may be expanded using 4.2 d) to give the

actual simulating net in a similar manner to that for R-paths.
(A description of R-processes first appeared in (Lauer and Campbell,1975) where the generalized Kleene star was not used.)

Procedure declaration.

Like E-processes, R-processes may exeeute procedures which have been given procedure bodies. A procedure body is introduced by a declaration given by the following production:-

P3: procedure ::= procedure identifier: (α)

where α is a regular expression.

The following restriction applies to the use of procedures with R-paths and R-processes:-

Restriction R3: No procedure named in a path expression may, during its execution, invoke by a series of other procedure calls another procedure named in that path expression.

Restriction R3 is similar to that of R3 in Chapter 4, and is a necessary condition imposed to prevent inappropriate procedure invocations leading to deadlock (see 4.10).

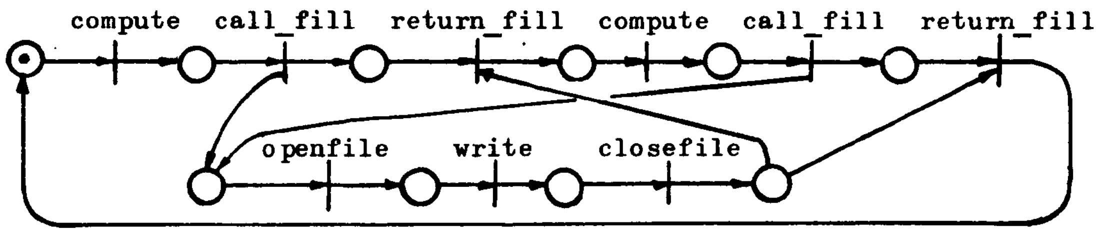
To conclude the discussion of this classification of R-processes we make the following remarks:-

R-processes are the first model of processes we have considered which involve a realistic model of procedures. The procedures of E-processes allow abstraction, operations on types, and hierarchically structured synchronization when used with paths. The procedure of an R-process may, in addition, be used to shorten the source text and the 'object code' (in the model, the simulating nets). For example:-

__process__ (compute ; fill ; compute ; fill)* __end__

__procedure__ fill: (openfile ; write ; closefile);

would generate the following simulating net using the procedure call mechanism of 4.1:-



Finding a criterion which would ensure freedom from deadlock for R-processes is more difficult than for E-processes. Let Ro-paths and Ro-processes be R-paths and R-processes respectively, constructed from a regular expression (α)* where α contains no selections or repetitions. Then there is no criterion analogous to C4 of Chapter 4 which ensures freedom from deadlock (4.6) unless repeated names within individual paths and processes are forbidden. Combining the simulating nets of Ro-paths and Ro-processes using the intersection transformation does not in general lead to a Marked Graph simulating net. (The intersection transformation introduces shared places when repeated names occur.) Of course, Ro-paths and Ro-processes are Eo-paths and Eo-processes if repeated names are forbidden. Hence, we must examine other Petri net classes and theorems if we require a criterion for freedom from deadlock for R-paths and R-processes.

5.5 IMPLEMENTATIONS OF R-PATHS.

R-paths cannot be implemented in the simple manner described for E-paths using a P operation in the prologue of the synchronized procedure and a V operation in the epilogue. In Chapter 3 we demonstrated that restrictions were necessary on a regular expression synchronization notation in order to allow an implementation in this manner, and showed that the restrictions imposed in the E-path notation were sufficient. In this chapter, instead of restricting the notation, we shall consider other implementation mechanisms.

The implementation mechanism for R-paths must incorporate the state machine nature of the R-paths within it in a more complex form than that for E-paths. One such mechanism is to utilize the Petri net hardware described by Patil (Patil, 1975) to implement the simulating net of the R-path directly. The other alternatives we suggest are based upon using the state machine representation of the R-path expression. The first of these alternatives that we shall suggest introduces a primitive operation in terms of which R-paths may be simply implemented. Another solution is to incorporate the state machine representing the regular expression of the path in a controller mechanism which is consulted on procedure entry and exit.

Implementation of R-paths using P,V selector operations on semaphores.

This, the first of the two alternative implementations we suggest for R-paths, is a method which introduces two primitive operations P' and V' which have as arguments sets of semaphores. Informally, the implementation scheme is based on representing the states of the state machine corresponding to the R-path by semaphores, as in the implementation for E-paths given in Chapter 3. However, a given procedure execution may be accepted from one of several possible states of the state machine, and it may leave the state machine in one of possibly many states. A process requesting to execute a given procedure will execute in the prologue to that procedure a P' operation on all the semaphores corresponding to the states from which that procedure execution may be accepted. The P' operation delays the process until it may decrement any one of the set of semaphores. Given the state from which a given procedure execution is accepted, there is a state in which acceptance of the procedure execution will leave the state machine. The semaphore which corresponds to this state is incremented by the process executing a V' operation in the epilogue of that procedure. The P' operation provides an index indicating the state responsible for the executien of the procedure, and hence P' can be regarded as a function. The V' operation uses this index to select the appropriate semaphore to increment from amongst the set of possible semaphores associated with the destination states of the transitions involving this procedure. Hence, the V' operation has an extra argument, i.e. that of the value of this index.

The P' or <u>P selector operation</u> will be written:-

P'(s1,s2,...,sn)

and will have the value v_j.

The V' or <u>V selector operation</u> will be written:-

V'(v_j,s1',s2',...,sm')

where the value of V_j indicates which semaphore from the set s1',s2',...,sm' is to be incremented.

Synchronized procedures will be implemented in the following form:-

```
procedure procedurename:
     begin  (comment start of prologue)
     index semaphore_to_be_freed;
     semaphore_to_be_freed := P(s1,s2,...,sn);
            (comment end of prologue)
```

```
                    .  .  .
            (body of procedure)
                    .  .  .

            (comment start of epilogue)
        V(semaphore_to_be_freed,s1',s2',...sn');
            (comment end of epilogue)
        end of procedure;
```

(For simplicity, we have omitted any parameters that the procedure might have.) s1-sn and s1'-sn' are semaphores, and semaphore_to_be_freed is an index which records the value of the index of the semaphore decremented by the P' operation. This index is then used to increment the appropriate semaphore in the V' operation. For illustration purposes we shall abbreviate the implementation to:-

```
P'(s1,s2,...,sn)   procedurename   V'(s1',s2',...,sn')
```

The appropriate P' and V' operations and semaphores for an implementation of an R-path expression can be generated from the deterministic state machine corresponding to the regular expression of the R-path (constructed by 2.19) using the informal description we have given. (Later, we shall give a precise definition of the P' and V' operations using the simulating nets of the R-paths.) An implementation can be constructed by using the following algorithm:-

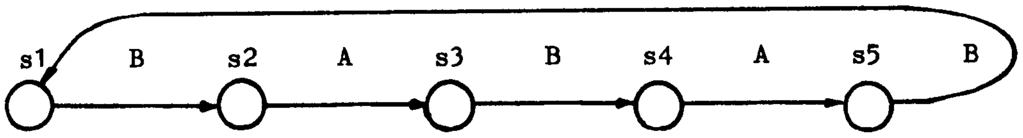
1) Declare and associate a semaphore with each state of the state machine.
2) Give the semaphore corresponding to the initial state the initial value one, and give all other semaphores the initial value zero.
3) A given procedure name may be used for the name of several

transitions in a state machine. The P selector operation used for that procedure includes, as arguments, all the semaphores representing states from which these transitions may occur. To each state which allows one of these transitions to occur, there will be a state to which the state machine is taken as a result of the transition occurring. The semaphores representing these states are included in the V selector operation in an order determined by the P selector arguments.

Example. Consider the following R-path:-

path (B ; A ; B ; A ; B)* end

This has the state machine given below, the states have been labelled with the names of the semaphores that will represent them.



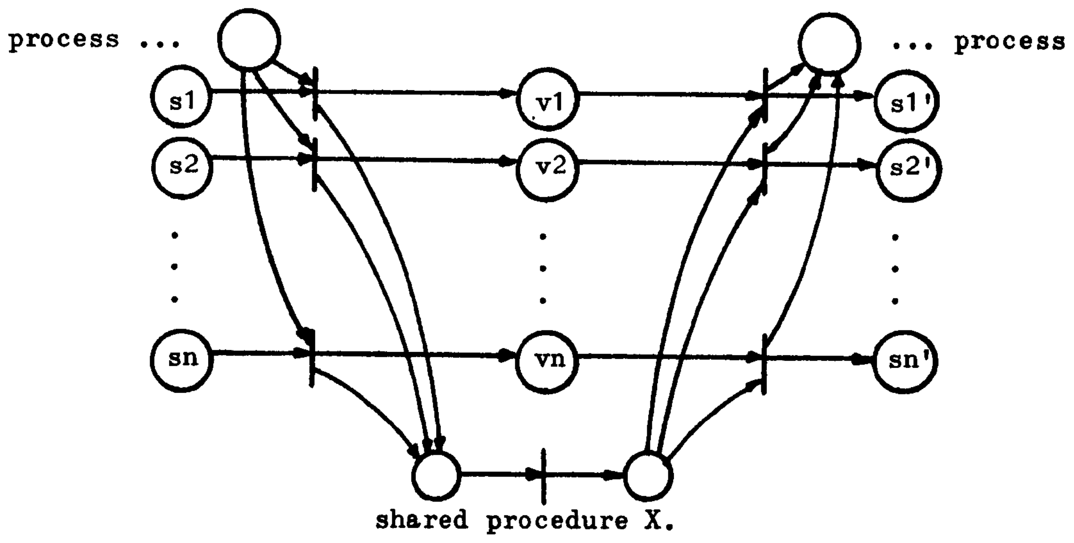
The implementation of the procedures is as follows:-

```
semaphore s1,s2,s3,s4,s5;
s1:=1; s2:=s3:=s4:=s5:=0;
P'(s1,s3,s5)          B          V'(s2,s4,s1)
P'(s2,s4)             A          V'(s3,s5)
```

A 'B' may be executed first, decrementing s1 and incrementing s2. This permits an 'A' to occur, decrementing s2 and incrementing s3.

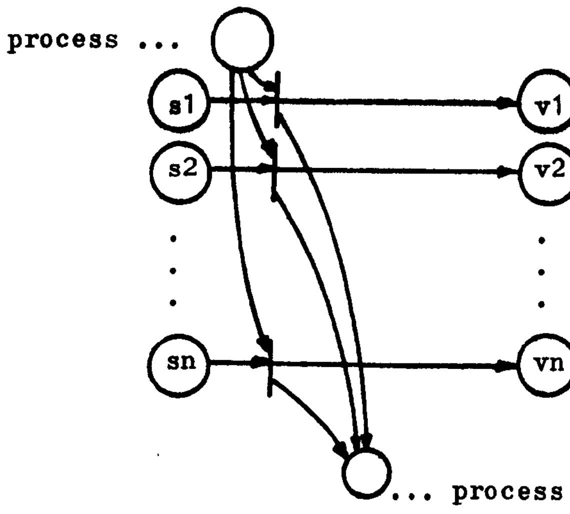
Definition of P and V selector operations.

We shall define the operations P' and V' by means of our model of processes and R-path expressions. The operations will correspond to subnets of a simulating net containing shared procedures synchronized by R-path expressions. A general characterization of such a shared procedure is given below and is derived from the intersection transformation TR5.1 after the abbreviations for shared procedure calls (see abbreviation 4.2 d)) have been expanded. To simplify the following description of the operations we shall label the places of Petri nets.

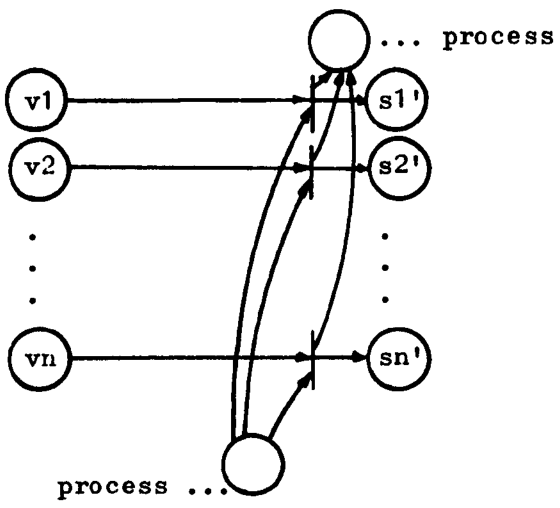


The primitive operations are subnets of this net.

<u>The P selector operator</u> (written P(s1,s2,...,sn) and with value v_j) <u>is defined as</u>:-



The V selector operation (written $V(v_j, s1', s2', \ldots, sn')$)
is defined to be:-



Having defined these two operations, we shall now give a characterization of them in terms of programming constructs. Later, we shall indicate how they might be implemented.

The places s1-sn and s1'-sn' and the transitions they are connected to behave in a similar fashion to semaphores and P, V operations. (See the Petri net representation of a semaphore given by Lautenbach (4.5).)
The P selector operation is a selection of a P operation (represented by the transition with input place si) from amongst a set of such P operations on semaphores. The selection is made arbitrarily between all those transitions with input places belonging to the set s1-sn and which are enabled. The additional freedom that a process may execute a synchronized procedure under one of possibly many conditions requires an additional implementation mechanism. This mechanism will be in the form of a scheduler as it is a predicate on the timings of processes (Lipton, 1973). In addition to selecting a P operation on some semaphore, the P' operation records which semaphore was decremented by marking a place v_j. This corresponds to the P selector operation returning the index of the semaphore decremented as a

value to the process which invoked it.

The V selector operation is a selection of a V operation represented by the transition with output place sj' from amongst a set of such V operations on semaphores. The selection is predetermined by the place vj containing a token. This corresponds to an index into the set of semaphores which has the value vj.

Implementation of P,V selector operations.

A simple mechanization of P,V selector operations can be made using a list structure, a critical section, P and V operations (Dijkstra, 1968b), and a scheduling scheme. The list structure associates with each P,V selector operation 'semaphore' a queue of items representing requests for P' operations to be performed on that 'semaphore'. The suspended P selector operation on a set of 'semaphores' is represented by a P' operation request in the queue of each of these 'semaphores'. Processes are actually suspended using their own private semaphores and the locations of these are also included in the items on the queues for the 'semaphores'. The critical section is used to synchronize modifications to the list structure. A scheduling decision must be taken when a V' operation increments a 'semaphore' upon which several processes are waiting and hence there is a choice of which process to release from its waiting state.

An alternative mechanization of P,V selector operations could be based upon a hardware implementation of these operations using the Petri net representation of them.

Implementation of R-paths using P,V operations on counting semaphores directly.

This scheme is suggested as an alternative to the one above, and implements the controller mechanism for R-paths directly in terms of counting semaphores, P,V operations and a representation of the automaton corresponding to the State Machine Petri simulating net of the R-path. In the previous implementation, R-paths were implemented by P,V selector operations which were in turn implemented by P,V operations on the private semaphores of processes. The controller implements the synchronization of processes by P,V operations on semaphores corresponding to the procedures which the processes wish to invoke.

A set of variables is associated with each R-path to represent the automaton and record state information. These variables are introduced by declarations which we give in the following Algol-like programming language:-

```
integer array     (1::n,1::j) table;
semaphore array   (1::j)S;
integer state;
semaphore mutex;
```

The table is a variable containing the state table representation of the automaton. The columns of the table correspond to the transitions which may occur in the automaton, the rows to the states. The automaton is presumed to have n states and j transitions. For example, suppose we had the R-path:-

path(A ; A ; (B , D) ; B)*end

then we would have n=4 and j=3 and initialization code to write the following matrix into the table.

transitions	table(i,1) (A)	table(i,2) (B)	table(i,3) (D)
states			
table(1,k)	2	0	0
table(2,k)	3	0	0
table(3,k)	0	4	4
table(4,k)	0	1	0

For example, in state 3 the transitions with non-zero entries in the row table(3,k) correspond to procedures which may be allowed to be executed by processes. Thus, a B or a D may be executed. When a transition (procedure) occurs from state 3, the resulting new state is given by the appropriate non-zero entry in table(3,k).

The semaphore array includes a semaphore for each procedure and provides the mechanism to allow processes requesting a procedure to wait. The integer state contains the current state of the automaton, held as an index from 1 to n.

State changes are implemented in the prologues and epilogues of procedures. During the execution of a procedure the state is given the value zero. The prologue of a synchronized procedure is as follows:-

```
prologue

    begin integer next_state; integer constant action;
    P(mutex);
    next_state: = if state = 0 then 0 else table (state,action);
    if next_state = 0 then
        begin   V(mutex);
                P(s(action));
                next_state:= table (state,action);
        end;
    state:=0;
    V(mutex);
```

The value of action is the index into the table of the column corresponding to that procedure. Next_state is a local variable of the procedure and records between the prologue and epilogue the value of the next state. The process is not allowed to continue until the value of next_state is non-zero.

The epilogue uses a scheduling procedure 'select' which returns from the table, semaphore array and state, the index of a procedure which may now be allowed to execute. 'Select' may use the identity of the procedures invoked by processes in order to choose between continuing the execution of one process as against another. This is different from the scheduling in the previous implementation where the scheduling operation could only choose between processes waiting for the same 'semaphore', and any given 'semaphore' might control the synchronization of more than one procedure. If there are no processes waiting to execute a procedure which may be accepted in the given state, 'select' returns a zero.

```
epilogue
        begin integer  next_action;
        P(mutex);
        state:=next_state;
        next_action:=select (table,S,state);
        if next_action=0 then V(mutex)
                          else V(S(next action));
        end;
    end of procedure;
```

Next_action is an index into the columns of the table and is a temporary variable used to hold the result of 'select'. If 'select' finds that there is a process waiting to execute a procedure which is permitted to execute by the new value of state, then that process is released to execute the procedure by the appropriate V operation. The released process is left executing in the critical section of a prologue. If there is no process to be released by the epilogue, then a V(mutex) is performed to allow other processes to enter the critical section.

Example: Suppose in the earlier example that state has the value 1. Suppose a process invokes B. The process will enter the prologue of B and read the entry table(1,2) with value 0 into next_state. Hence, the process will suspend itself by executing P(S(2)). Suppose a process invokes D. The process will enter the prologue of D, set its next_state variable to the value of table(1,3) which is 0, and suspend itself by executing P(S(3)).

Suppose a process invokes A. The process will enter the prologue of D, set its next state variable to table (1,3) which has a value of 0, and suspend itself by executing P(S(3)).

Suppose a process invokes A. The process will enter the prologue of A, set its next_state variable to table (1,1) which has a value of 2, set state to zero, and proceed to execute the body of A.

Suppose another process invokes A. The process will enter the prologue of A, find the value of state is zero, and suspend itself by executing P(S(1)).

Eventually, the process executing the body of A will enter the epilogue and set state to the value of next_state, a 2. The select procedure has only one procedure to choose from. This is the procedure of A which has been invoked since S(1)=-1. Hence select returns the value 1 and the process executing the epilogue now executes a V(S(1)), releasing the process waiting to execute a second A. The released process sets its next_state to table (2,1) which has the value 3, sets state to zero, and executes the body of A.

Eventually, it will execute the epilogue of A, set state to 3, and call select. This time select may choose between returning a 2 or a 3, depending upon its scheduling policy.

We have given several implementations with the intention of illustrating that R-paths are sufficiently general in nature to be implementable on many different computing systems. The P, V selector operations described above allow the implementation of the R-paths to be undertaken at the process level of description, that is we activate or suspend individual processes by the mechanism. An alternative is to implement the R-paths at the machine level by including hardware to represent the simulating nets of the paths. Finally, we have given an implementation at the level of procedure invocations. The first implementation requires a global list structure and critical section in which to manipulate it. The last implementation given could be integrated very efficiently and effectively with a machine architecture which supports procedure calls in a manner of the B6700 and could be the topic of further research. A further implementation consideration is the trade-off between making the compilation efficient, and reducing space in the program compiled. We shall not consider these aspects in depth, but shall merely point out that there exist many techniques to reduce the size of a state machine and that these techniques may be applied to R-paths. Thus, many implementations of R-paths can be envisaged, and it is reasonable to expect that the method adopted in any given practical implementation will be tailored to suit the characteristics of the underlying computing system.

5.6 A COMPARISON OF R-PATHS AND E-PATHS

As a direct measure of the declarative power of synchronization of R-paths we may use the expressive power of regular expressions and of state machines. Hence, we conclude that R-paths have more declarative power than E-paths. The scheduling power of the R-path notation is at least that of the E-path notation since any E-path expression is equivalently an R-path expression. The scheduling power of the E-path notation was shown to be that of a semaphore notation in 4.6. This semaphore notation could be used in the implementation of an R-path (5.5) in place of the binary semaphore since no V operation is performed on a semaphore with value 1. However, from Theorem 3.4, although the E-path notation resolves conflicts between sets of sequences of actions it imposes the restriction that the predicate upon which a procedure is permitted to be executed must depend upon only one synchronization condition. The R-path notation allows a procedure to be executed by a process under one of several different synchronization constraints. This results in an inability to express conflicts in an R-path solely in terms of conflicts in E-paths (or the semaphore notation of 4.6) and additional programming in the form of a scheduler must be introduced to resolve conflicts. We, therefore, conclude that R-paths are more powerful than E-paths with respect to scheduling power.

Finally, we shall make some remarks about the conversion of R-paths to E-paths. A given R-path may not be in its simplest form, and hence it may not be immediately obvious whether it is equivalent to some E-path or not. However, the state machine corresponding to this R-path may be minimized and by 2.20 the resulting minimized state machine is unique up to isomorphism with other minimized state machines with the same behaviour. However, from Chapter 3 we have the Lemma 3.3 which states that the state machine constructed for an E-path by 3.1 is a minimum state machine. Hence, proving equivalence between an R-path and an E-path is a trivial matter, For the construction of an E-path from a given R-path we may minimize the state machine of the R-path and then construct an appropriate regular expression which is equivalent to this state machine and which is acceptable to the set of E-path productions, using construction rules based upon 3.1 and 2.19 used in reverse. Of course, not all R-paths have equivalent E-paths.

5.7 FURTHER EXAMPLES OF R-PATHS.

Synchronous Processes. (Bekkers, 1974).

In this example we have two processes. At one stage of execution, each process must wait for the other process so that the two processes can restart together. For this particular solution, we will assume that the processes are not aware of their separate identities. In our illustration we shall declare a type called synchro which will incorporate the correct synchronization.

```
    type synchro;


        path (arrive ; arrive ; depart ; depart)*end;
        procedure depart: ( body of depart);
        procedure arrive: ( body of arrive);
        procedure synk:   ( arrive ; depart) end;
    operations synk
    endtype
```

A process invoking synk will execute arrive and then wait to execute depart. It will be able to proceed only after a second process has executed arrive. The example illustrates the use to which the unrestricted Regular Expression nature of an R-path may be put. A similar type including only E-path expressions would require additional variables or that the processes are aware of their separate identity (see 4.8). Each procedure arrive and depart may occur under two different synchronization constraints (that is, as the first occurrence of a request for that procedure or as the second).

File handler.

The next example concerns file handling. A process may request the use of a file to either just read, or just write, or to both read and write it. Once the request has been granted, the file cannot grant access to another process until the first relinquishes its facility to read or write. Requests are performed by opening the file; requests are terminated by closing the file. Once opened, the file may be used by many processes to read or to write. Mutual exclusion prevents synchronization errors in reading or writing. The file operates in one of three modes depending on the access request:-

reading, a write, and reading and writing arbitrarily intermixed.

```
type file;


    path        (( ropen : read*; rclose) , (wopen; write; wclose),
                (rwopen ; (read , write)*; wclose)) *end
    procedure   read: (read routine);
    procedure   write: (write routine);
operations ropen, rclose, wopen, wclose, rwopen, rwclose,
                bufferopen, bufferclose
endtype
```

We leave the open and close actions undefined for brevity; they might, of course, have null bodies.

The type file allows reading and writing to occur in a number of different synchronization constraints. The single R-path is sufficient to allow all these synchronization constraints to be expressed without recourse to anyadditional programming.

R-paths used to implement co-ordinated but parallel execution of actions.

The R-path mechanism can be used to implement synchronization and co-ordination between a group of parallel processes invoking operations on data in which several of the operations may be executed simultaneously, but in mutual exclusion to other operations.

For example, let us suppose that operaticn A,B and C are to be synchronized so that execution of an A occurs concurrently with the execution of the procedure B. However, the operations A and B are mutually exclusive with the execution of a C. The following type definition provides this synchronization:-

```
type X;
        path((start_A ; start_B),(start_B; start_A) ;
                    (finish_A ; finish_B) , (finish_B ;
                    finish_A) ; C)*end;
        procedure A:  ( start_A ; body_of_A ; finish_A );
        procedure B:  ( start_B ; body_of_B ; finish_B );
        procedure C:  ( body_of_C );
        operations    A, B, C
endtype;
```

The type definition synchronizes only the beginnings and endings of the procedures A and B, allowing concurrent execution of their bodies. In general, this technique for specifying possible concurrent execution of procedures subject to other synchronization constraints results in large R-path expressions which are not readily comprehensible. In the next chapter we shall consider an extension to E-paths and R-paths which permits a clearer statement of such forms of concurrency.

5.8 SUMMARY OF R-PATHS.

We have generalized the notion of a path notation based on regular expressions given in Chapter 3 and found that the generalization provides more flexibility and power to the programmer. The resulting R-path expressions were shown to have implementations using P and V operations on semaphores. The implementations are, however, more complex than that for E-paths. The extension of E-paths to R-paths required a definition of the meaning of the repetition of a procedure name within a path. We also extended our classification of processes to include a more realistic notion of a procedure and demonstrated that this new class of processes, together with R-paths represented as simulating nets, required a more sophisticated approach to its analysis than that described in Chapter 4 for E-paths and processes.

Finally, we noted in the last example that path expressions could be used in type definitions to synchronize the occurrence of concurrent events. This provides the motivation for the extension to path expressions considered in the next Chapter.

CHAPTER 6.

GENERAL PATH EXPRESSIONS

To conclude the examination of path notations based upon regular expressions, we shall examine a notation in which several regular expressions may simultaneously describe the synchronization and co-ordination of an action. In the previous chapters we imposed the restriction that a given procedure name may occur in only one path expression. In this chapter we shall examine the effect of relaxing this restriction and of placing a particular interpretation upon the repetition of an action in different paths. We shall show that this notation is more flexible and has greater declarative power than the previous notations and that it has a significant effect upon structuring synchronization specifications of actions. We shall also investigate the implementations of this notation and compare them with the implementations for E-paths and R-paths.

Whereas E-paths and R-paths allow an action to be synchronized with respect to a sequence of actions, the General Path (or G-path) notation permits an action to be synchronized with respect to several independent and concurrent sequences of actions. A General Path expression will comprise of several E-paths or R-paths which have procedure names in common. The occurrence of a given procedure name in several path expressions will imply that the synchronization conditions specified in each path for that procedure must hold simultaneously for that procedure to become eligible for execution by a process. Unlike E-paths and R-paths, a G-path may include several procedures which, although synchronized with respect to other procedures, may be executed concurrently with respect to each other.

As in the previous chapters, we shall describe the meaning of G-paths with respect to processes using a Petri net representation and describe various implementations. We shall illustrate the structuring that G-paths permit for the description of synchronization problems. A further sub-class of processes is introduced which permits the examination of processes which share procedures, and properties are examined of the combination of such processes and G-paths. Finally, the advantages and disadvantages of this notation over that of E-paths and R-paths are examined.

6.1 GENERAL PATH NOTATION.

A general path expression (G-path) is a collection of E-path expressions and R-path expressions in which the names of actions occur in more than one path. The R-paths and E-paths are defined by the description in Chapter 5 and Chapter 3, respectively, ignoring however the restriction which prevents the repetition of procedure names between path expressions. A G-path permits a procedure to be executed if all the path expressions in which that procedure name occurs allow that procedure to be executed. For example:-

The G-path,

```
        path  (A ; B ; A)*end
        path  (A ; C)*end
```

synchronizes the procedure A so that it may only be executed by processes if both path expressions allow it to be executed. Hence, if two processes invoke A, after the first process has executed A, the second process must wait until both the procedures B and C have been executed by other processes before it may execute A.

The simulating net for the G-path is constructed from the simulating nets of the constituent E-paths and R-paths by a transformation given below. The transformation ensures that all the synchronization conditions in all the appropriate paths must hold before a given action may occur:-

TR6.1 <u>Transformation to generate the abbreviated simulating net of a G-path</u>.

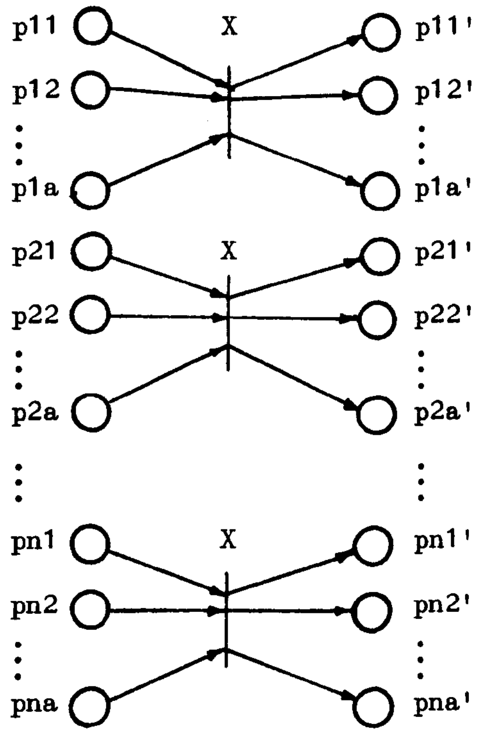
1) Generate the abbreviated simulating nets of any component R-paths (5.2) and the simulating nets of any component E-paths (4.1).

2) Select any simulating net of a path expression in the G-path and call it the primary net. Replace this primary net with a new primary net by selecting one of the remaining simulating nets of the component path expressions of the G-path and performing the following construction:-

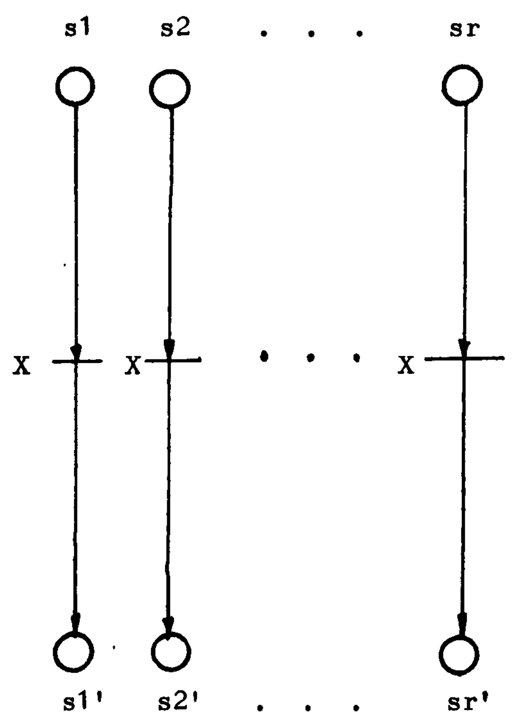
3) For each unique procedure name which occurs both in the primary net and in the simulating net do 4).

4) Identify all the transitions in the primary net which are labelled with that name (suppose there are n such transitions).
Identify all the transitions in the simulating net which are labelled with that name (suppose there are r such transitions).
Transform these transitions in the following manner:-

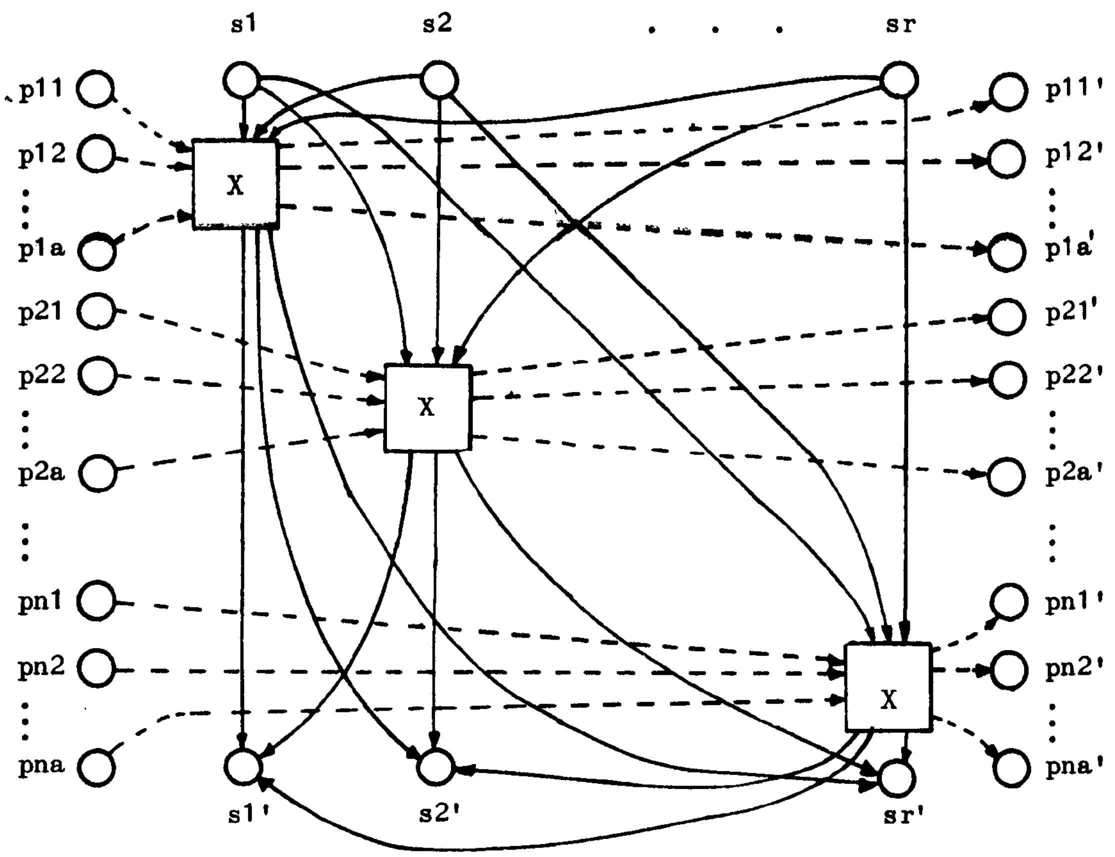
Transitions with label X from primary net.



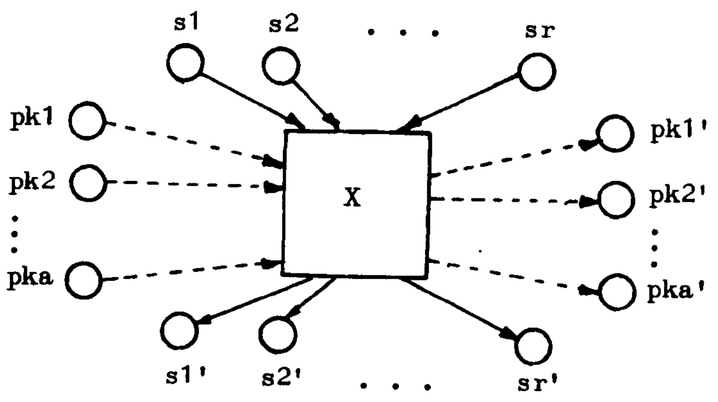
Transitions with label X from simulating net.



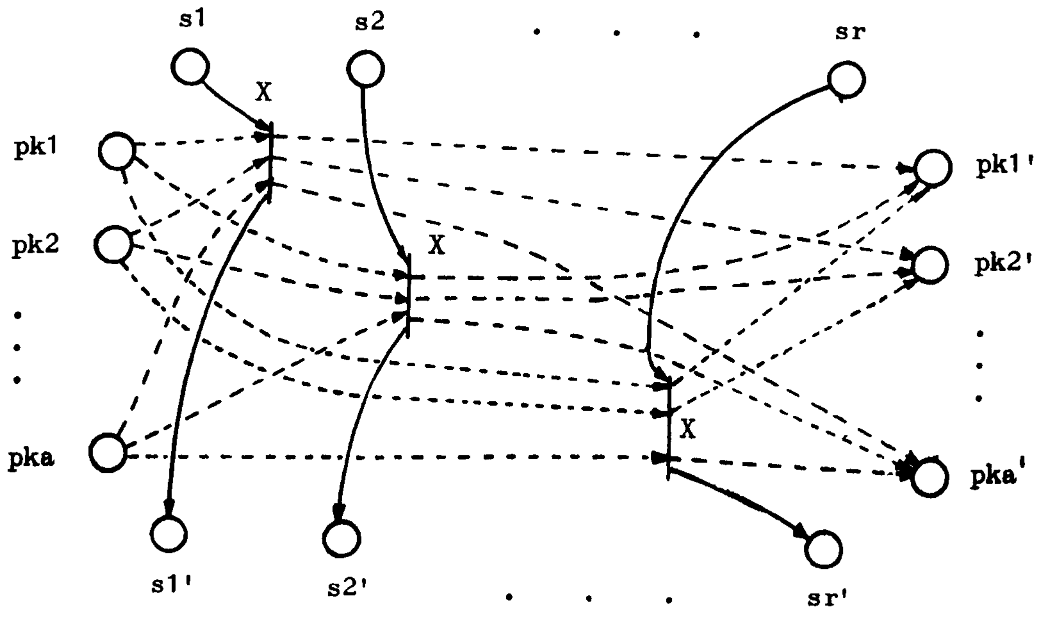
by:-



where the box:-



is an abbreviation for:-



(1 ≤ k ≤ n).

(The number of transitions generated for a given procedure name at each application of the above transformation will be the product of the number of times the transition occurred in the simulating net, and the number of times it occurred in the primary net. Thus, in the diagram above, the transformation has created n x r transitions labelled X. Each of these transitions is an abbreviation for the calling mechanism to a single transition labelled X which represents the actual execution by a process of the procedure X (see 5.2).

The number of input and output arcs for each transition in the original primary net is increased by one corresponding, in the new primary net, to the extra constraint imposed by the addition of another path. Thus, in the above diagram, X has 'a' input arcs in the original primary net, and a+1 input arcs in the new primary net.)

5) While there are simulating nets of component path expressions of the G-path which have not been combined with the primary net, replace the present primary net with a new one constructed by applying 3) to one of these simulating nets and the present primary net.

6) The final primary net is the abbreviated simulating net of the G-path.

   To obtain the simulating net for the G-path the abbreviated simulating net is expanded (see 5.2).

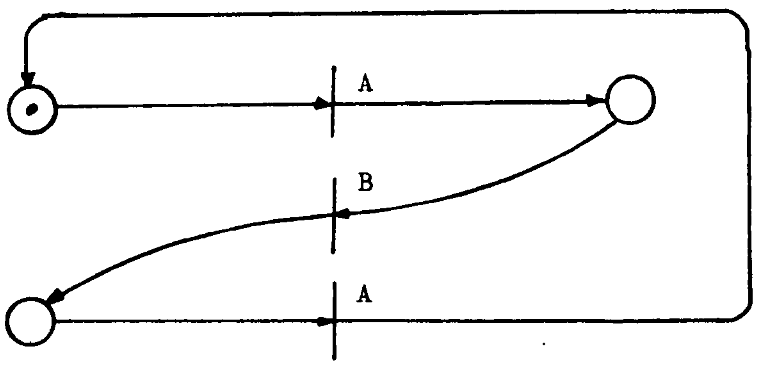
<u>Examples</u>.

Consider the following G-path:-

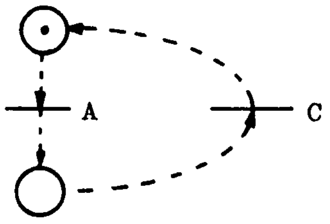
<u>path</u> (A ; B ; A)* <u>end</u>

<u>path</u> (A ; C)* <u>end</u>

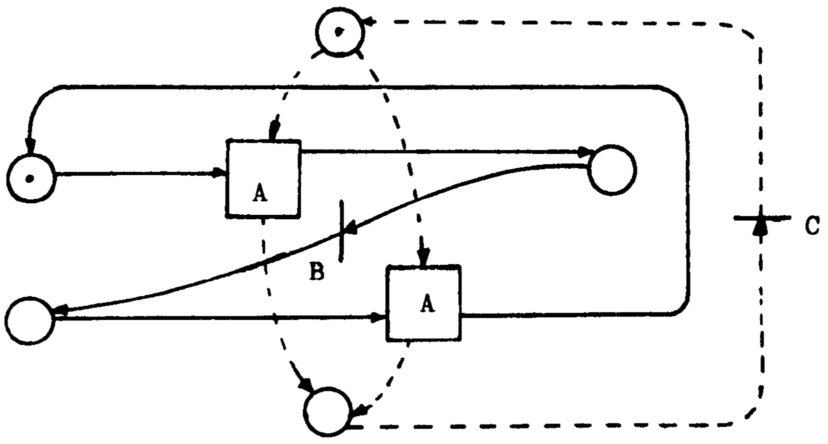
Let the simulating net of the first path be the primary net. This net is as follows:-



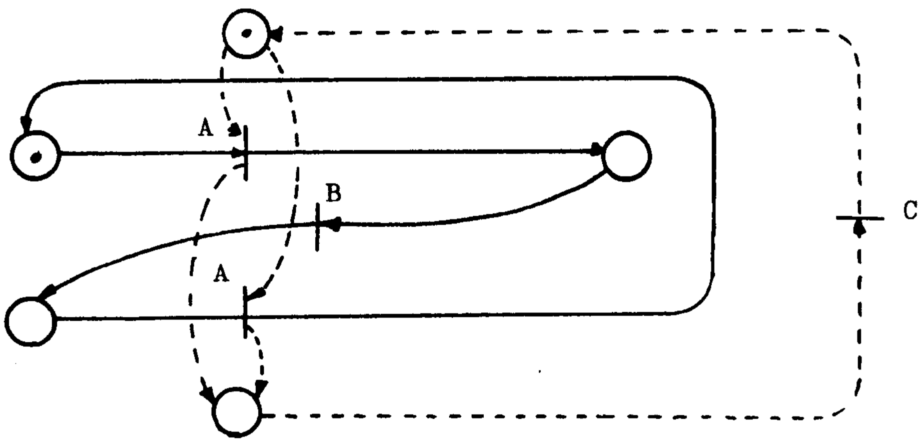
The simulating net of the second path expression (distinguished by dashed arcs) is:-



Combining the two nets using the transformation yields:-

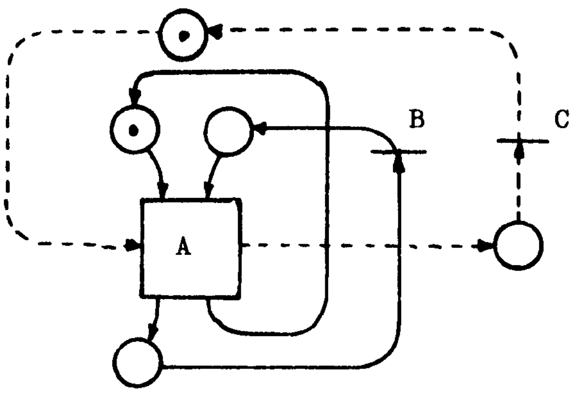


where the boxes are abbreviations. If we expand the boxes we obtain the simulating net for the G-path:-



From the abbreviated simulating net of the G-path it may be seen that the transitions B and C may both occur at the same time. This corresponds to procedure B and C both being executed simultaneously by separate processes.

The order in which the simulating nets are taken in the construction does not affect the resulting simulating net. If, instead of the first net, we choose the second net as the primary net then, re-arranging the layout of the nets a little, the result of combining the two nets is:-



Expanding the box abbreviation and again re-arranging the layout of the nets a little gives the simulating net shown above.

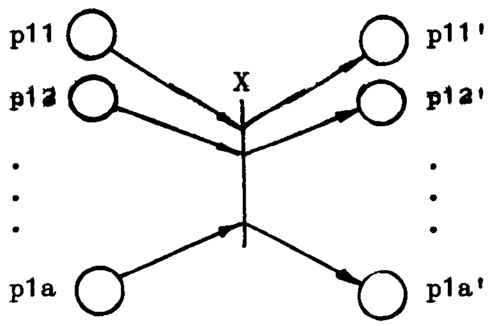
The transformation can be simplified if the G-path is composed solely of E-paths. <u>Let a GE-path be a G-path composed</u>

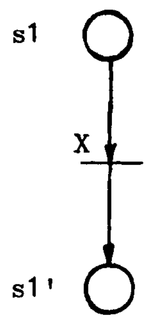
only of E-paths. By Lemma 3.3 the simulating net of an E-path will have only one transition labelled with a given procedure name. Consider part 4) of the transformation. Whenever a primary net is being combined with the simulating net for an E-path, r equals 1. Assume that the initial primary net is chosen to be a simulating net of an E-path, then n equals 1. Given any primary net such that it has only one transition labelled with a given procedure name then if it is combined with the simulating net of an E-path, the transformation will result in n X r transitions with the same name, i.e., one transition. Hence, by induction the transformation applied to the simulating nets of the component paths of a GE-path will always generate a new primary net in which only one transition is labelled with any given procedure name. Thus, 4) of the transformation can be simplified to:-

Replace:-

Transition with label X from primary net.

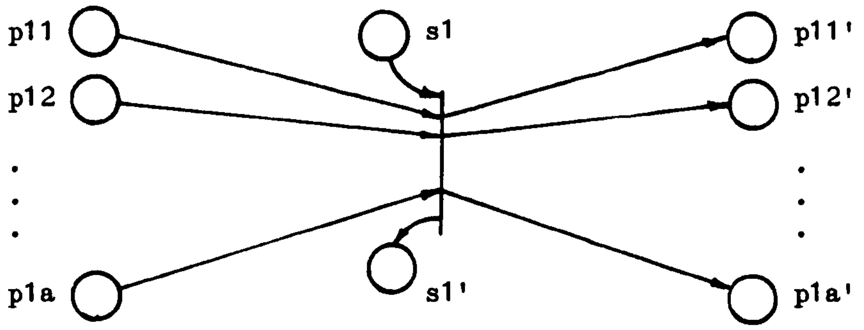
Transition with label X from simulating net.





by:-

Transition with label X in new primary net.



The following comparison can be made between G-paths and GE-paths:-

A procedure synchronized by a GE-path may be executed by a process if all of a set of conditions hold.

A procedure synchronized by a G-path may be executed by a process if all of a set of conditions hold in one of the elements of a set of sets of conditions.

Examples of G-path Expressions.

The G-path expressions allow synchronization specifications which are too complex to be written simply as an R-path or an E-path expression. The following examples illustrate the synchronization which may be expressed.

The Cigarette Smokers Problem of Patil.

Patil (Patil, 1971) introduced the following synchronization problem:-

"Three smokers are sitting at a table. One of them has tobacco, another has cigarette papers, and the third has matches; each one has a different ingredient required to make and smoke a cigarette but he may not give an ingredient to another. On the table in front of them, two of the three ingredients will be placed, and the smoker who has the necessary third ingredient should pick the ingredients from the table, make a cigarette and smoke it." Further ingredients are not put on the table until the old ones have been consumed. Other smokers must not interfere with the smoker who has the ingredients on the table before him. Hence co-ordination is required between the smokers.

The cigarette smokers problem can be restated (Lauer and

Campbell, 1975) in the following way:-

1) Decide which of the ingredients should be put on the table.

2) Produce each ingredient and place it on the table.

3) Choose the correct consumer to consume the available ingredient.

4) Go back to 1).

Where 1) is an arbitrary choice from producing:-

1.1) tobacco and a match.("supplytm")

1.2) a match and paper. ("supplymp")

1.3) paper and tobacco. ("supplypt")

Where 2) is determined from the decision made in 1) in the following way:-

2.1) given "supplytm" or "supplypt" produce tobacco.

2.2) given "supplymp" or "supplytm" produce a match.

2.3) given "supplypt" or "supplymp" produce paper.

Where the choice 3) is determined upon 2) such that:-

3.1) tobacco is consumed by either "matches-smoker" or "paper-smoker".

3.2) paper is consumed by either "tobacco-smoker" or "matches-smoker".

3.3) a match is consumed by either "paper-smoker" or "tobacco-smoker".

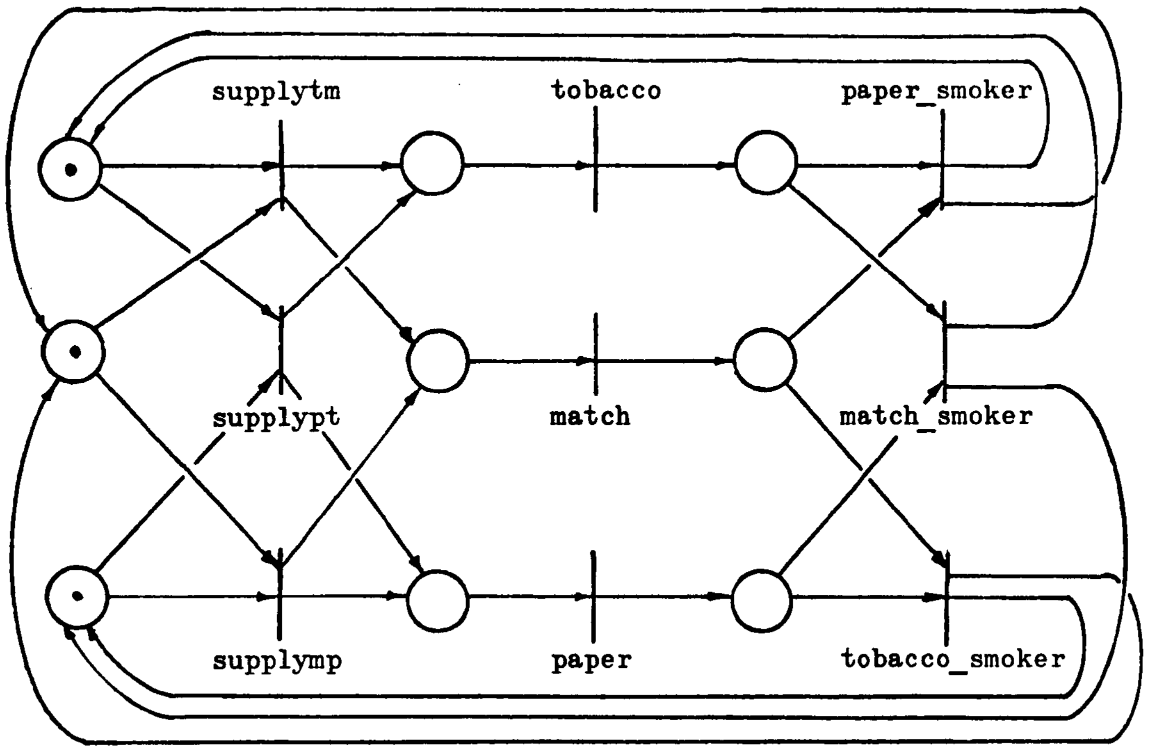
The ingredients tobacco, match and paper are manipulated by several processes and are shared objects. The decision to supply a given ingredient, to produce and to place it on the table, and to consume it are all operations on that ingredient. For each ingredient we may write a path expression which co-ordinates the execution by processes of these operations. For example, for the ingredient tobacco we may write:-

```
path ((supplytm,supplypt) ; tobacco ; (matches_smoker,
      paper_smoker)) *end
```

The following GE-path describes the synchronization and co-ordination of the Cigarette Smoker's problem; each separate component path expression of the GE-path describes how the operations on a particular ingredient may occur:-

```
path ((supplytm, supplypt) ; tobacco ; (matches_smoker,
     paper_smoker))*end
path ((supplytm,supplymp) ; match ; (tobacco_smoker,
     paper_smoker))* end
path ((supplypt,supplymp) ; paper ; (tobacco_smoker,
     matches_smoker))*end
```

Although the GE-path is structured into sequential components (the E-paths), it states the synchronization required for a problem in which concurrency occurs. Each E-path component describes the synchronization with respect to a particular shared object, the GE-path describes the combined synchronization properties of those objects treated as a composite shared object. The simulating net for the GE-path is as follows:-



(Note that this net is almost identical to the Petri net definition of the problem given by Patil (Patil, 1971).)

An Example based on the Five Dining Philosophers of Dijkstra

The manner in which GE-paths allow separate components of a synchronization specification to be combined avoids some of the problems described in Chapter 4 on hierarchies of procedure calls involving path expressions. The following GE-path (which is similar to the Five Dining Philosophers problem of Dijkstra (Dijkstra, 1973)) synchronizes five procedures P0-P4 so that only two of them may be executed concurrently at any one time. The GE-path permits a procedure to be executed if its neighbouring procedures are not being executed. That is, it will permit the execution of procedure Pi if neither procedure P(i+1 modulo 5) nor P(i-1 modulo 5) are being executed by processes. Thus, for example, P1 may be executed if P0 and P2 are not being executed, and P1 and P3 may be executed concurrently by processes provided no process is executing any of the procedures P0,P2 and P4.

```
path (P0,P1)* end

path (P1,P2)* end

path (P2,P3)* end

path (P3,P4)* end

path (P4,P0)* end
```

(Each component E-path expression of the GE path corresponds to the synchronization specification required to allow a fork in the Five Dining Philosophers problem to be used by only one Philosopher at a time.)

This synchronization may be written as a set of disjoint E-paths which are hierarchically structured by procedure calls. However, this synchronization specification may lead to deadlock, as in the following example. A process first invokes a procedure Pii which then invokes Pi. If five processes simultaneously invoke and execute the five procedures P00-P44, deadlock will arise as they proceed to invoke P0-P4 (see the last remark of 4.10).

```
path (P00 , P1)* end

path (P11 , P2)* end

path (P22 , P3)* end

path (P33 , P4)* end

path (P44 , P0)* end
```

Similarly, non-deadlocking hierarchical structures of this kind lead to giving a priority to processes executing one procedure over those executing others. For example:-

```
path (P0 , P1)*  end

path (P11 , P22)* end

path (P2 , P33)*  end
```

path (P3 , P44)* end

path (P4 , P00)* end

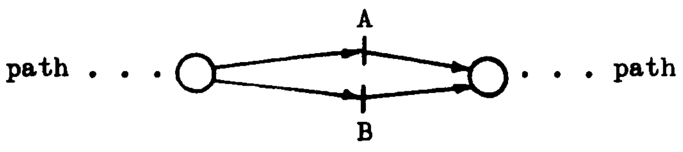
where Pii invokes Pi.

If all the Pii procedures are invoked simultaneously by separate processes , then P33,P44,P00, and one of P11 or P22 may be executed concurrently. However, the process executing P33 cannot execute P3 because another process is executing P44. Similarly the process executing P44 cannot execute P4 because another process is executing P00. If the process invoking P11 is allowed to execute P11, then the process invoking P22 will be delayed. The process executing P11 will invoke P1. However, the process executing P00 will invoke P0 and executions of P0 and P1 are mutually exclusive. Thus, in this case, only one process is permitted to execute a procedure from the procedures P0-P4, whereas the original specification allowed two. Similar problems arise with other configurations of procedures and paths (see Dijkstra, 1973).

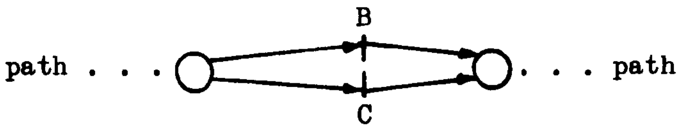
Thus, from the examples above, we conclude that GE-paths allow a given synchronization problem to be structured into sequential components (as in the first two examples) which are combined automatically and correctly, as opposed to the difficulties that arise (which was illustrated in the last two examples) when sequential components are combined by means of hierarchically structured procedure calls. Finally, we believe that the GE-path notation allows complex problems of synchronization to be expressed clearly and simply.

6.2 PROPERTIES OF THE SIMULATING NETS OF G-PATHS

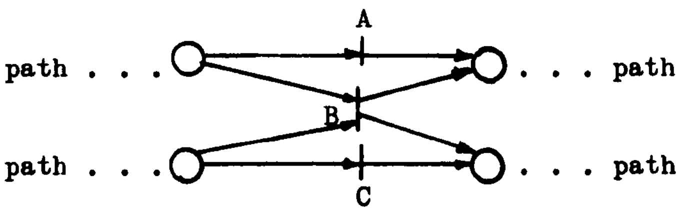
As in earlier chapters for R-paths and E-paths, we shall examine the properties of G-paths by investigating their simulating nets (G-nets). R-paths generate simulating nets which, in their unabbreviated form, belong to the Simple Petri net class (Lemma 5.1). G-paths, however, do not belong to this classification because transformation TR6.1 permits a transition to have more than one shared input or output place, contrary to the definition of a Simple net (2.10). For example, suppose the procedures A, B and C are in component paths of a GE-path and are in simulating nets:-



and



The G-net formed by TR6.1 from these nets will have a subnet:-



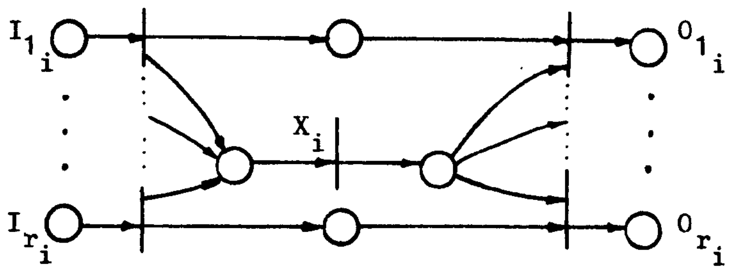
Simple Petri nets are the largest sub-class of nets contained within General Petri nets described by Holt and Commoner (Holt and Commoner, 1970), and hence we must classify the simulating nets of G-paths as belonging to General Petri nets if we use their classification scheme. The transformation TR6.1 generates G-nets from the nets of R-paths or E-paths, and hence an abbreviated G-net is decomposable into safe State Machine Petri nets. Thus, although G-nets belong to a classification of nets which is more general than Simple Petri nets, they belong to a sub-class of General Petri nets.

Theorem 6.1: G-nets are safe.

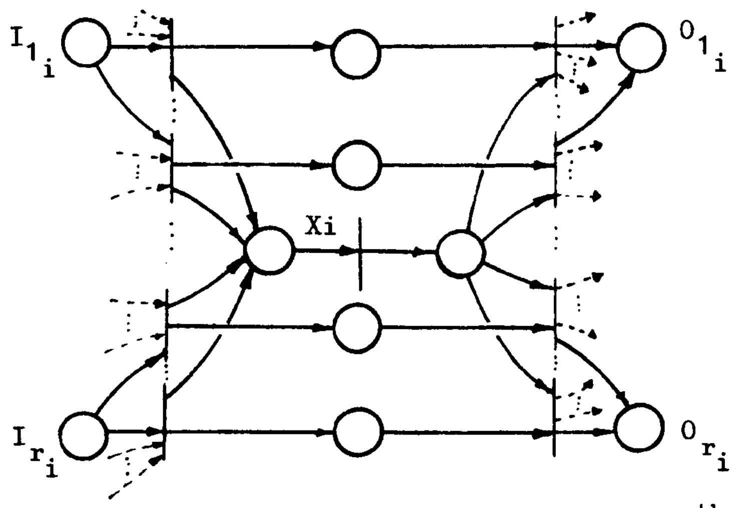
Proof: In general, a G-path will be constructed from a set of component R-paths (the set of regular expressions which may be written within an E-path expression is a subset of the regular expressions which may be written within an R-path). From Theorem 5.3, the unabbreviated simulating net of an R-path is safe and is a Simple Petri net (lemma 5.1). Hence the first primary net generated by the transformation (TR6.1) is safe in its unabbreviated form. We will prove the theorem by induction on the number of primary nets that are generated during the transformation.

Suppose, as induction hypothesis, that the qth primary net generated is safe in its unabbreviated form. Suppose that we have a simulating net of an R-path which has not yet been combined with a primary net (if there is no such net the transformation terminates).

Let X_i be the ith unique procedure name which labels r_i transitions in the abbreviated simulating net of this R-path. Before applying part 3 of TR6.1, the unabbreviated simulating net of this R-path will contain a subnet:-



The places are labelled for convenience, the dashes represent connections to the rest of the simulating net. Since the net is safe only one token can mark any of $I_{1_i} - I_{r_i}$ at any one moment. No further token can mark $I_{1_i}-I_{r_i}$, once marked, until the terminal transitions from $O_{1_i} - O_{r_i}$ have removed a token from one of $O_{1_i} - O_{r_i}$ (note this applies even if a place O_{j_i} is also labelled I_{n_i}). After applying 3, the subnet will be as follows (the dots represent connections to the rest of the q+1th primary net).



(We suppose that the procedure name X_i occurs in the qth primary net as well as in the simulating net otherwise the second net would be identical to the first and there is nothing to prove. The number of subnets representing the calling mechanism (5.2) to X_i will be increased, but they are not all shown. If the qth primary net had n_i such calling mechanisms for the transition labelled X_i, then the new one will have $n_i \times r_i$ (see TR6.1).)

Every initial transition of the input place of transition X_i above has an input place belonging to the set $I_{1_i} - I_{r_i}$. Marking an input place I_{k_i} will result in the eventual marking of place O_{k_i}, as in the original subnets. If I_{k_i} is marked then no other token can mark one of $I_{1_i} - I_{r_i}$ until O_{k_i} has one of its terminal transitions fire. Hence the q + 1th unabbreviated primary net is safe. Hence, by induction, on application of part 3 of TR6.1, the net constructed by TR6.1 is safe in its unabbreviated form. Q.E.D.

The safeness of G-nets indicates that, although in the earlier examples we demonstrated that G-paths may describe actions occurring concurrently, an individual procedure described by a

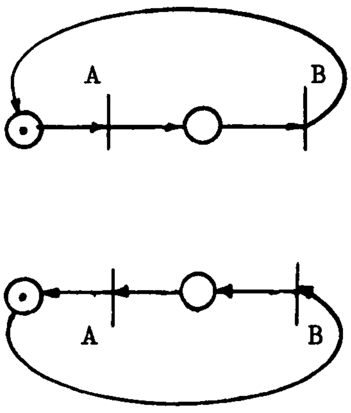
G-path cannot be executed simultaneously by several processes.

G-path expressions do, however, allow synchronization expressions to be written which are deadlocked or may result in deadlock. Consider the following GE-path:-

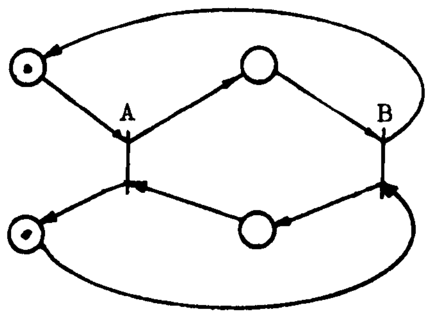
path A ; B end

path B ; A end

The component E-paths have simulating nets which are individually live. However, when TR6.1 is applied to these nets, the resulting simulating net for the GE-path has no transitions which are live:-



becomes under TR6.1:-



The deadlock can also be recognized by a theorem of Petri net theory. The simulating net for the GE-path is a Marked Graph (2.9) and includes an empty cycle . Therefore, we conclude (from 2.12) that not all the transitions of the net will be live. In Chapter 4 we gave a criterion C4 which ensured that a set of Eo-paths and Eo-processes were deadlock free. We may write a similar restriction to ensure that a sub-classof the GE-path notation cannot include a deadlock, (see criterion C5, and Theorem 6.6 on page 164.)

6.3 A MODEL FOR A PROGRAM CONSISTING OF PROCESSES AND G-PATHS.

The model we shall give for a program consisting of G-paths and processes is based upon our earlier models in 4.3

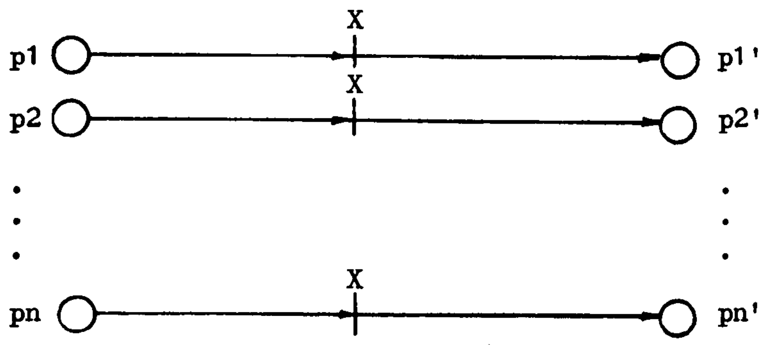
and 5.3. The following intersection transformation gives the abbreviated simulating net for a program consisting of G-paths and processes.

TR6.2 <u>Intersection transformation for the nets of processes and G-paths</u>.

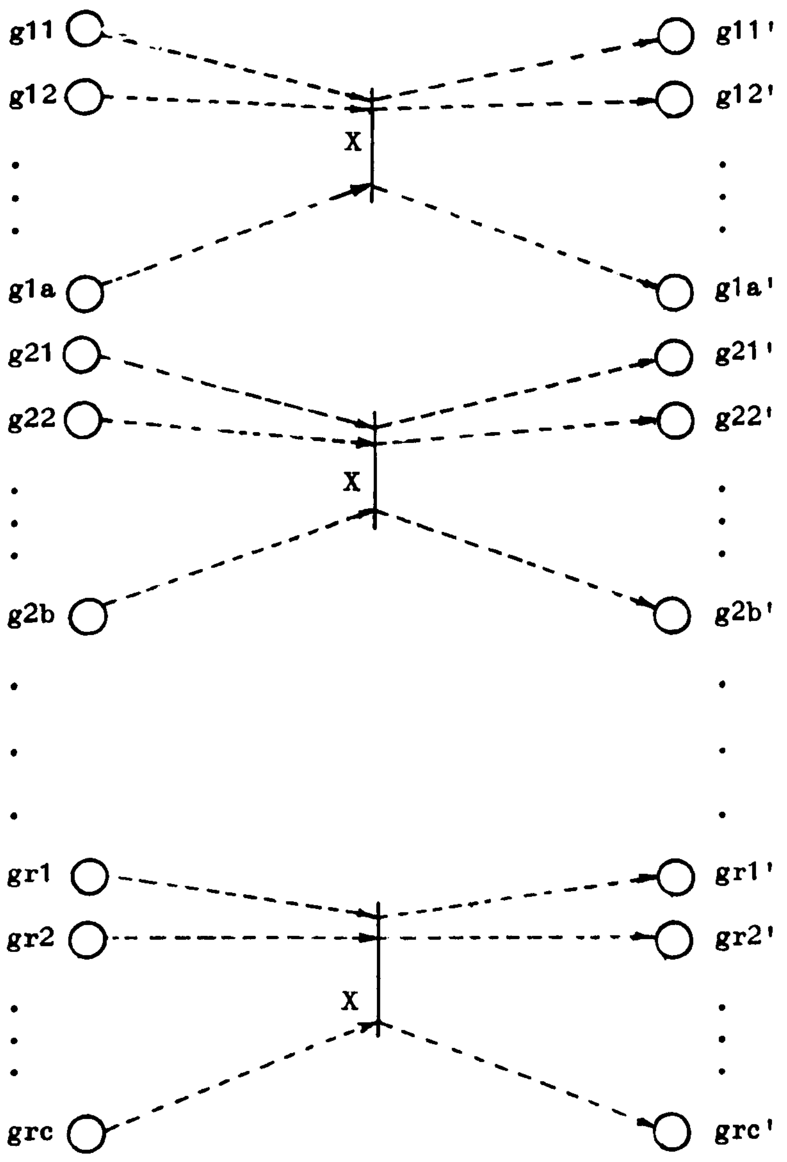
Abbreviate the simulating nets of the G-paths (5.2). A given action or procedure name may label several transitions in a G-net. (Unlike TR5.1, these transitions may have several input and output places.) For each unique name do the following:-

1) Identify the set of transitions in the abbreviated G-net with that name.
2) If, in the process nets, the name corresponds to the body of a procedure embedded in the procedure calling mechanism, then abbreviate the calls upon that procedure by replacing them by transitions labelled with that name using abbreviation 4.2 d). Identify the set of transitions with that name.
3) Apply the following transformations to the set of transitions identified.

Replace the subnet:-



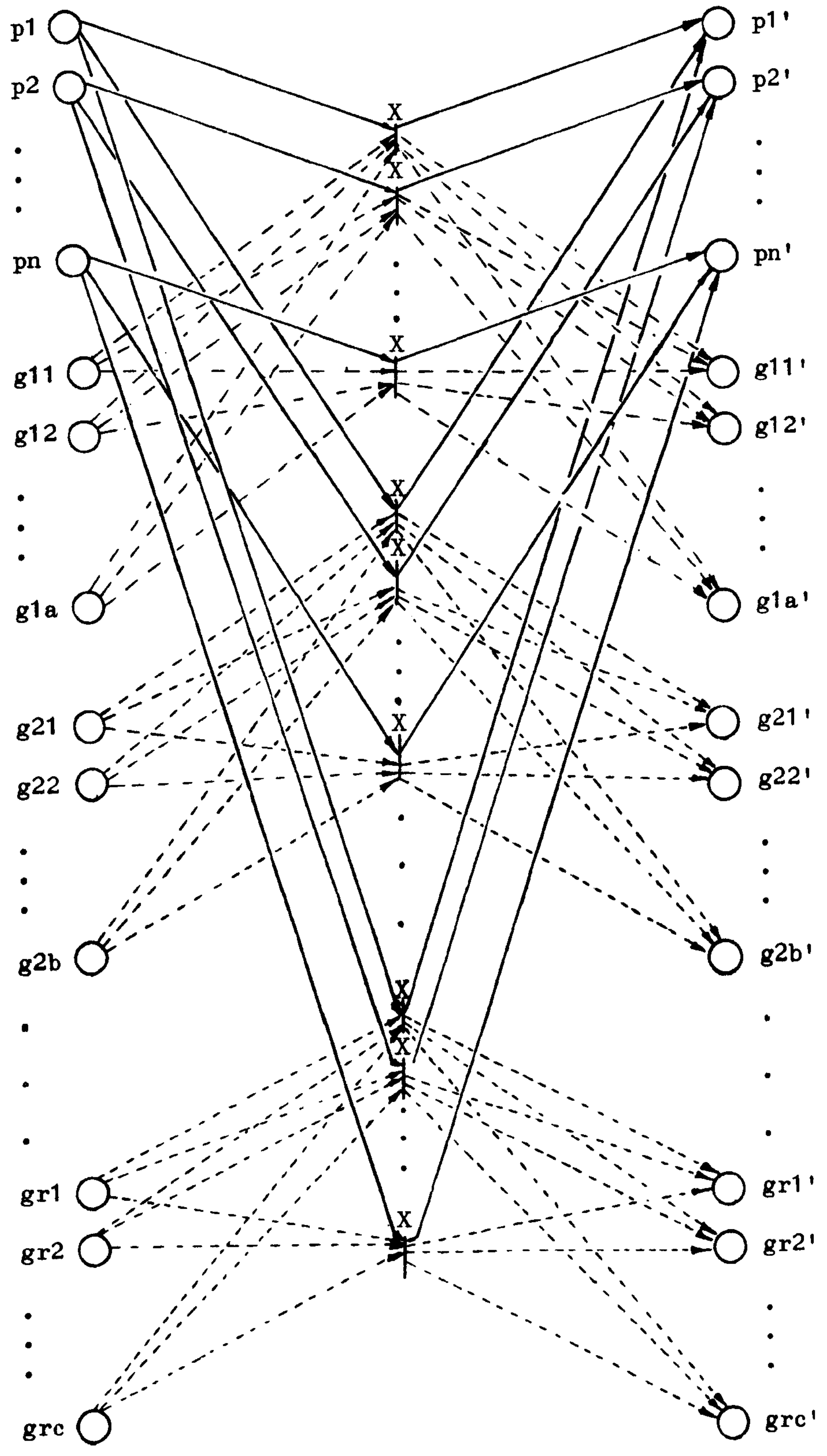
n transitions belonging to the Petri net model of the processes.



r transitions belonging to the simulating net of the G-path.

(Subnets belonging to the G-net are distinguished by dashed arcs.)

by the subnet:-



n × r transitions belonging to the resulting Petri net model of the program consisting of G-paths and processes.

The places of the subnets involved (for clarity the places are labelled in the diagrams above) are not affected by the transformation although they may be given more input or output arcs to transitions labelled X. Each process of the n processes is connected to the G-path through arcs to and from the n x r transitions. The n x r transitions are all labelled with the same name and their firing corresponds to the execution of the procedure X. Thus, the transitions we have introduced are abbreviations (4.2 d) for the calling mechanism to a unique transition labelled X.

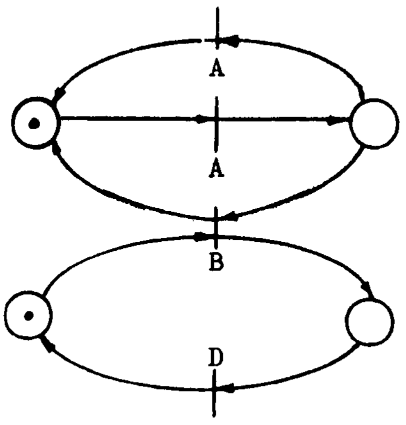
<u>Example of the intersection transformation between a G-net and a Petri net model of a program.</u>

The following G-path

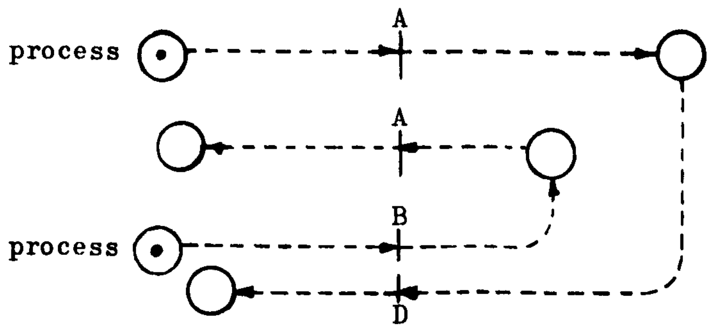
<u>path</u> (A ; (A , B))* <u>end</u>

<u>path</u> (B ; D)* <u>end</u>

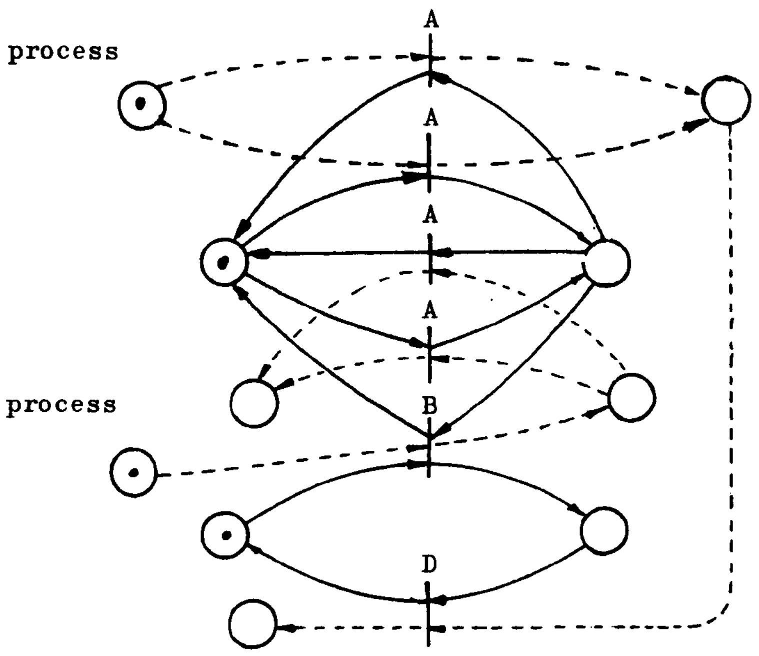
has the simulating net (abbreviated):-



Suppose we have a set of processes (distinguished by dashed arcs):-



Then combining the processes with the G-path using TR6.2 gives:-



where the dashed arcs correspond to the subnet representing the processes. The final net contains four transitions labelled A and these represent abbreviations for the calling mechanism of (4.2 d) to a single transition labelled A. The net permits the A and B to be executed in sequence followed by possibly simultaneous execution of D and A.

For a model of GE-paths and processes, TR6.2 becomes much simpler. In this case, the number (r) of transitions labelled with an identical procedure name in the abbreviated GE-net will be at most one (see TR6.1, the simplified transformation to generate the simulating net for a GE-path). Suppose part 3 of 6.2 is applied to the procedure name X and there are n transitions in the process net representing abbreviations to the procedure calling mechanism of X. The result of the transformation specified in 3 will be a subnet containing n x r transitions (i.e n transitions since r=1) representing abbreviations for the calling mechanism. Thus,the number of transitions representing procedure calls to X in the process model will remain unchanged in the model for the program consisting of those processes and GE-paths.

6.4 A G-PROCESS CLASSIFICATION OF PROCESSES.

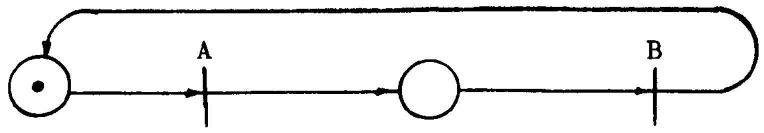
As in 5.4 and 4.6 we may introduce a G-process as a counterpart to a G-path. A G-process consists of a set of R-processes or E-processes in which procedure names are allowed to appear in more than one path. The occurrence of a given procedure in several processes is treated as if that procedure is shared between the processes and may be executed, perhaps concurrently, by the processes in which it occurs. The abbreviated simulating net for a G-process consists of the abbreviated simulating nets of the component processes. The simulating net for the G-process is obtained by removing the abbreviations using 4.2d). A G-process consisting only of E-processes will be called a GE-process.

Example. Consider the following G-process:-

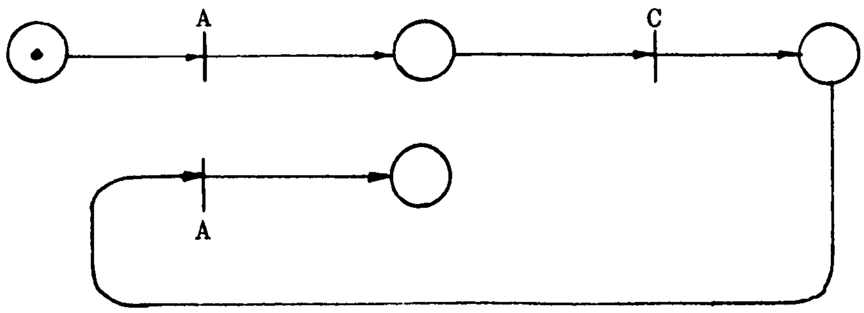
process (A ; B)* end

process A ; C ; A end

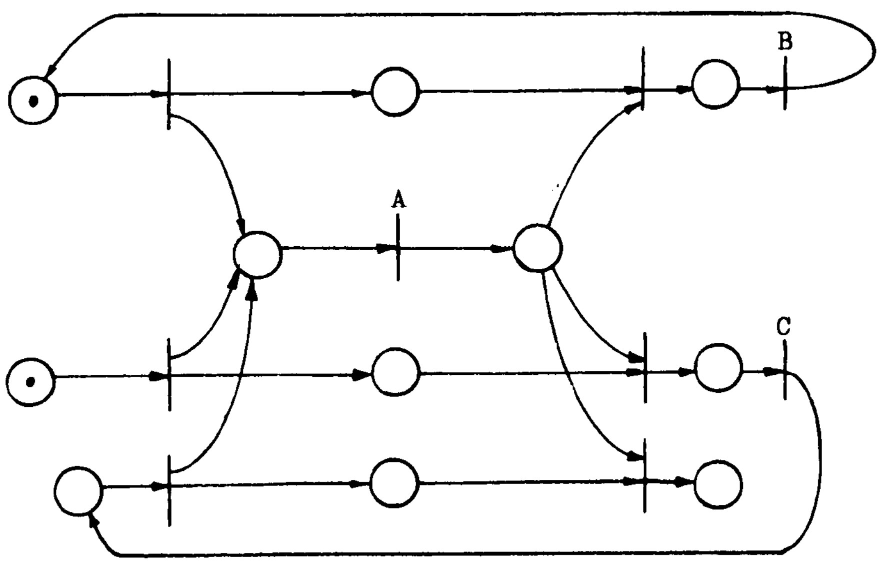
The processes have simulating nets which are respectively:-



and:-



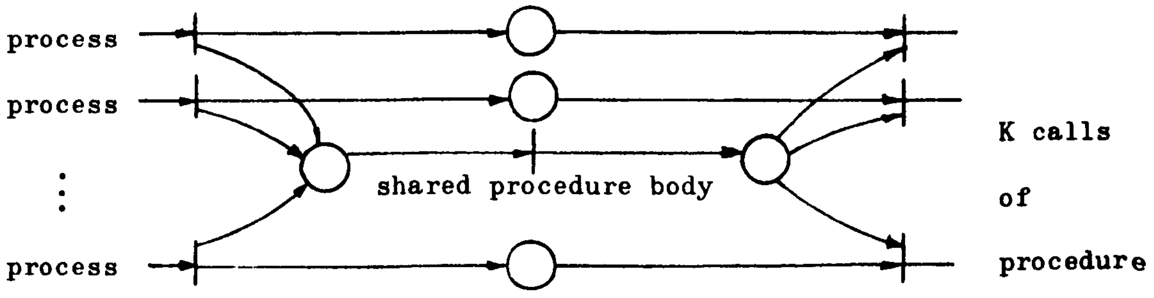
Removing the abbreviations gives:-



The processes may both execute A simultaneously and, in the net above, the input place of the transition labelled A is two safe (that is, it may have two tokens marking it). G-processes introduce procedures into our classification scheme which may be executed concurrently, and hence may be used to model problems of indeterminacy and data corruption.

<u>Theorem 6.2</u>: A G-process in which the maximum number of times any procedure is repeated between R-processes is N has a simulating net which is N safe (see 2.4 and 2.5).

<u>Proof</u>: Suppose a procedure name occurs in K different R-processes. There will be at least K procedure calls from those processes to the procedure body corresponding to that name:-



Each of the K processes may make a procedure call to the procedure body simultaneously, resulting in the input place to the procedure body containing K tokens. Thus, the safeness of the whole simulating net of a G-process corresponds to the safeness of the input place of the procedure which is shared by the largest number of processes.

Q.E.D.

The property of safeness can be used to investigate concurrency within a given program of G-processes and G-paths.

Example of an asynchronous ring buffer system.

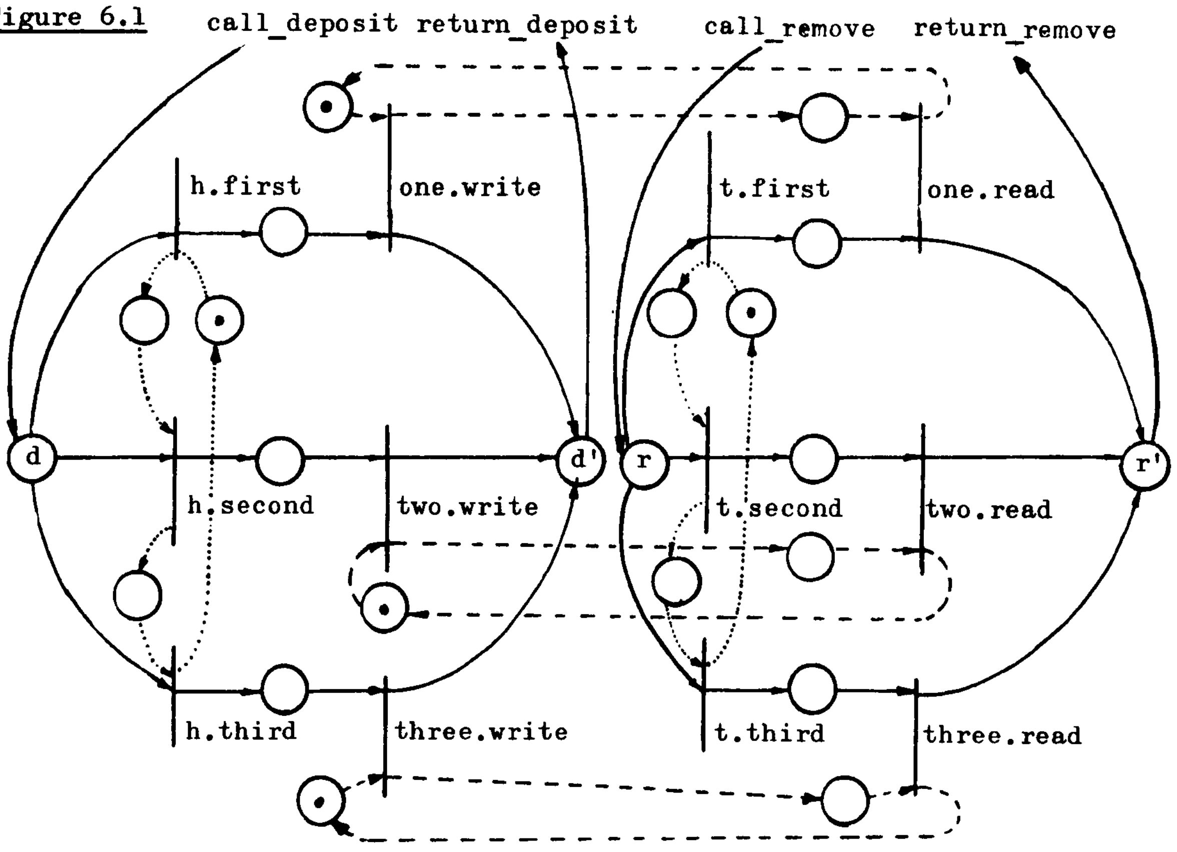
The following example is an extension of an example in 4.7 and is an equivalent expression of the synchronization required for an asynchronous ring buffer to that described in 1.2. The ring buffer system has three 'frames' which are used for the buffering and are allocated for use in a round robin fashion. There are two consumers and two producers. The program describing this system and its simulating net are shown in the following pages.

Ringbuffer example program.

```
begin
    type frame;
            path ( write ; read )* end
    operations read,write
    endtype;

    type allocator;
        path (first ; second ; third )* end;
    operations first, second,third
    endtype;

    type ringbuffer;
        frame    one,two,three;
        allocator h, t ;
        procedure deposit: ( (  h.first ; one.write ),
                             (  h.second ; two.write ),
                             (  h.third ;  three.write) );
        procedure remove:  ( (  t. first ; one. read ),
                             (  t.second ; two. read ),
                             (  t.third ; three. read) );
        operations remove, deposit
    endtype;
ringbuffer R;
process ( R.remove ; consume 1 )* end;
process ( R.remove ; consume 2 )* end;
process ( create 1 ; R. deposit )* end;
process ( create 2 ; R. deposit )* end;
end.
```

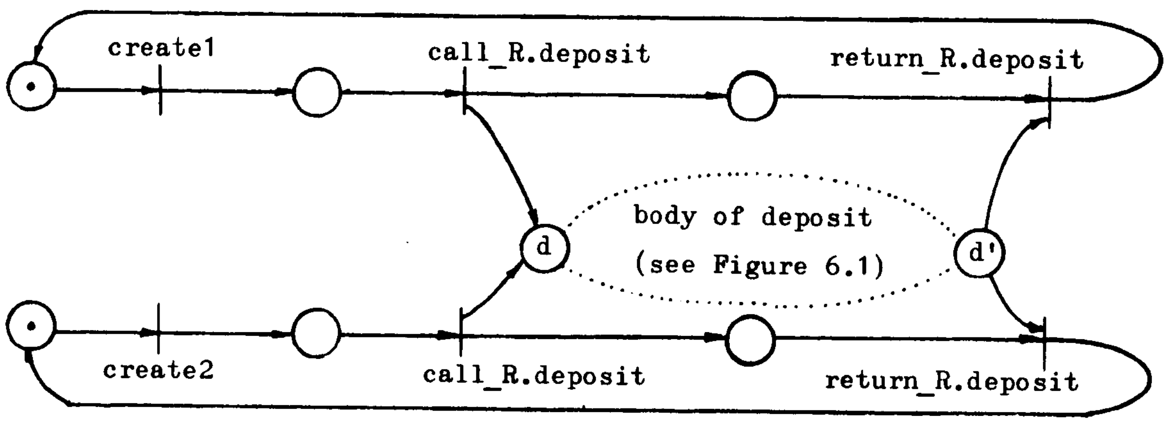
The type ringbuffer corresponds to the following simulating net:-

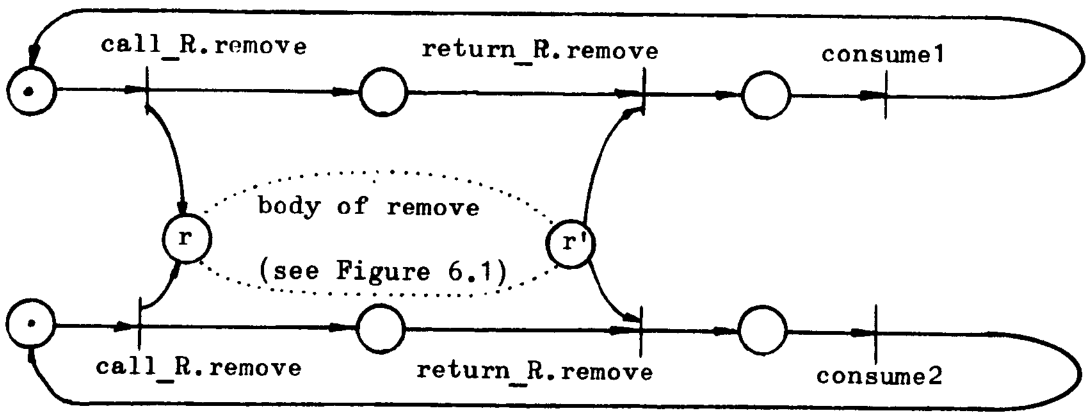
Figure 6.1



The operations deposit and remove are distinguished in the figure by solid arcs in their corresponding simulating nets. The operations deposit and remove may be requested concurrently, and their input and output places, labelled respectively d,d' and r,r', may not be safe. Thus the type must be examined to find its behaviour under such circumstances. The subnets representing instances of type 'frame' are distinguished in the figure by dashed arcs. These subnets ensure that the operations read and write on an instance of 'frame' can only occur in mutual exclusion. Further, the subnets also ensure that a write must occur before a read. The subnets representing instances of the type 'allocator' are distinguished in the figure by dotted arcs. These subnets ensure that the operations first, second and third on an instance of 'allocator' can only occur in mutual exclusion. In addition, these subnets also ensure that the instances of 'frame' will be used in a round robin manner. The first process executing deposit must proceed by executing 'h. first' and then 'one.write'. The next two processes executing deposit will execute 'h.second' followed by 'two.write', and 'h.third' followed by 'three.write' respectively. Up to three processes may execute their respective 'writes' simultaneously. Similarly, up to three processes may be executing 'one.read', 'two.read',...., 'two.write', 'three.write'. Only one process may execute one of 'h.first', 'h.second' and 'h.third'. Similarly only one process may execute one of 't.first', 't.second', and 't.third'. Thus there may be up to five processes actually executing actions within the operations 'deposit' and 'remove' simultaneously. However, many more processes may concurrently be executing these two operations.

Finally, 'remove' and 'deposit' occur twice in different processes and hence the shared procedures 'deposit' and 'remove' will be two safe, that is there may be two simultaneous executions of 'deposit' or two simultaneous executions of 'remove'. The simulating net of the processes is shown below where the bodies of 'R. deposit' and 'R. remove' are specified by the simulating net of the type definition of ringbuffer given in figure 6.1.





The example illustrates that our model of G-processes and G-paths may be used to describe complex problems. In Chapter 1 we described a ringbuffer which involved the use of a pointer, addition, an array and indexing of an array. However, in the example above, we have described a ringbuffer directly in terms of the actions which may occur. We believe the model we have given for processes and paths is a practical tool, and could provide the basis for future research and the study of the behaviour of operating systems.

<u>Criterion for freedom from deadlock for a set of G-processes and G-paths</u>.

The following example illustrates a deadlock occurring in a path expression and two processes:-

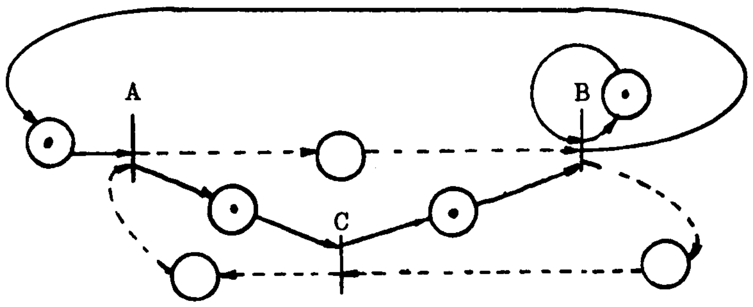
<u>path</u> (A ; B)* <u>end</u>;

<u>path</u> (B ; C)* <u>end</u>;

<u>process</u> (C ; A)* <u>end</u>;

<u>process</u> (B)* <u>end</u>;

None of the transitions in the following simulating net for these paths and processes are live:-



The simulating net above is a Marked Graph (2.9) and includes an empty cycle indicated by the dashed arcs. Hence the net is not live by 2.12. In Chapter 4 we gave a criterion (C 4) on Eo-paths and Eo processes which guaranteed freedom from deadlock. We may give a similar criterion for G-paths consisting of Eo-paths and G-processes consisting of disjoint Eo-processes (that is, the processes contain no shared procedures).

Criterion C5: Given a set of G-paths and G-processes such that each G-path consists of Eo-paths, and each G-process consists of disjoint Eo-processes, then this set cannot contain paths and processes in which there are procedures X1 to Xn forming sequences:-

```
...      X1      ...  ;  ...       X2       ...
...      X2      ...  ;  ...       X3       ...
          .                         .
          .                         .
          .                         .
...      Xn      ...  ;  ...       X1       ...
```

Theorem 6.6: The criterion C5 is a necessary and a sufficient condition for programs consisting of G-paths and G-processes to have a live and safe Marked Graph simulating net.

Proof: The proof of this theorem follows identical arguments to those given in Theorem 4.6. Inspection of TR6.1 and TR6.2 reveals that when they are applied to G-paths and G-processes which meet the criterion C5, these transformations are identical to those of TR4.3.

Q.E.D.

Thus we have a criterion for detecting deadlocks in G-paths and G-processes, although this criterion can only be applied in very limited circumstances.

6.5 IMPLEMENTATION.

The implementation of G-paths is much more complex than that of E-paths and R-paths. As with R-paths, given suitable hardware, the simulating nets of G-paths could be implemented directly. However, we shall describe three alternative implementations, two briefly and a third in more detail.

Implementation of G-paths using R-paths.

An implementation of G-paths can be given in terms of R-paths in the following manner. A procedure synchronized by a G-path will be implemented by replacing it with a procedure in the following form:-

```
procedure  procedurename:
           begin
           (comment start of prologue)
           start_procedurename;
           (comment end of procedurename)
           .   .   .
           (procedure body)
           .   .   .
```

```
(comment start of epilogue)
 finish_procedurename;
(comment end of epilogue)
end;
```

The procedure is rewritten with a prologue and an epilogue. The prologue includes an invocation of a procedure called start_procedurename, and the epilogue includes an invocation of a procedure called finish_procedurename. These two procedures are used to give the synchronization required for the original procedure. Executions of these two procedures are synchronized by an R-path. The R-path is constructed to permit processes to execute these procedures whenever the G-path synchronization specification would have allowed a process to execute the original procedure.

For example, consider the G-path:-

```
path (A ; B)* end;
path (A ; C)* end;
```

The procedures A,B, and C will be implemented in the form:-

```
procedure A:  begin  start_A ; (body of A) ; finish_A end;
procedure B:  begin  start_B ; (body of B) ; finish_B end;
procedure C:  begin  start_C ; (body of C) ; finish_C end;
path  ( start_A ; finish_A ;
                      ( start_B ; (finish_B ; start_C ; finish_C ),
                                  ( start_C ; (finish_B ; finish_C ),
                                              (finish_C ; finish_B ) ) ),
                      ( start_C ; ( finish_C ; start_B ; finish_B ),
                                  ( start_B; ( finish_C ; finish_B ),
                                              ( finish_B ; finish_C ) ) )
                                                                          ) *end;
```

together with declarations for the procedures start_A, start_B, ... , finish_C which will have null bodies.

The R-path which synchronizes the procedures in the implementation is constructed from the G-path in the following way:-

1) Replace each occurrence of a 'procedurename' in the G-path by the regular expression:-

   (start_procedurename ; finish_procedurename)

   Thus, for the G-path used in the example above we obtain:

   <u>path</u> ((start_A ; finish_A) ; (start_B ; finish_B))*<u>end</u>

   <u>path</u> ((start_A ; finish_A) ; (start_C ; finish_C))*<u>end</u>

   The new G-path and the procedures in their implementation form specify the same synchronization for the procedure bodies as that specified by the original G-path.

2) Generate the state machines corresponding to each component path expression of the G-path.

   Each state machine corresponding to a component path expression is extended to accept transitions corresponding to actions which are named in the other component path expressions of the new G-path, but which are not named in this component. Such transitions may occur in any state, and they return the state machine to the state from which they were accepted. The example G-path would generate two state machines as follows:-

First component of the G-path.

state	transitions					
	start_A	finish_A	start_B	finish_B	start_C	finish_C
1	2				1	1
2		3			2	2
3			4		3	3
4				1	4	4

(The state machine is extended to accept start_C and finish_C.)

Second component of the G-path

state	transitions					
	start_A	finish_A	start_C	finish_C	start_B	finish_B
1	2				1	1
2		3			2	2
3			4		3	3
4				1	4	4

(The state machine is extended to accept start_B and finish_B.)

3) We shall construct a new state machine which accepts a a sequence of actions if that sequence is also acceptable to all the state machines constructed by 2). This state machine and the procedures in their implementation form specify the same synchronization for the procedure bodies as that specified by the original G-path. However, the procedures which are invoked in the prologues and epilogues of the implemented form of the procedures are constrained by this new state machine specification of the synchronization so that they are executed by processes in mutual exclusion.

The new state machine is constructed, using a standard result from state machine theory, by taking the cross-product of all the state machines (Harrison, 1965) created in 2) above.

Thus, for the example G-path we would construct the following state machine:-

	transitions					
state	start_A	finish_A	start_B	finish_B	start_C	finish_C
11	22					
13					14	
14						11
22		33				
31			41			
33			43		34	
34			44			31
41				11		
43				13	44	
44				14		41

The unreachable states are not shown.

4) Construct a regular expression which has the same behaviour as the state machine given in 3). (An algorithm for one such construction is given by Harrison (Harrison, 1965).) The R-path which synchronizes the procedures in the implementation of the G-path is then given by embedding the regular expression between a <u>path</u> <u>end</u> bracket. Applying 4) to the state machine obtained for the example G-path gives the R-path given earlier.

This implementation may be used directly, as described above, or it may be adapted to give an implementation of G-paths using one of the R-path implementation techniques discussed in Chapter 5. This method of implementation has the advantage that, in a programming environment where there is a mixture of R-paths and G-paths, they may all be implemented using the same technique without incurring any overheads for R-paths by virtue of having the G-path facility. However, a G-path which permits parallel execution of the procedures that it synchronizes will generate, in general, a very large state machine cross-product and corresponding R-path. As yet the question of whether the size of such R-paths would be prohibitive and whether reduction techniques might be applied to reduce them to a more manageable size has not been investigated and could provide a topic for further research. Finally, we remark that procedures which cannot be executed by any process because of the synchronization specification in the G-paths

(for example, the GE-path:-

```
path (A ; B)* end
path (B ; A)* end
```

which has a simulating net which is not live) cannot be synchronized correctly using the above R-path implementation. Thus, such incorrect specifications must be detected prior to or during the application of the constructions.

Implementation of G-paths using the incidence matrices of the abbreviated simulating nets and P, V operations on counting semaphores.

This scheme is suggested as an alternative to the one above and has the advantage of being compact. In a similar manner to the implementation for R-paths using P, V operations on counting semaphores in Chapter 5, we shall introduce a controller which co-ordinates the executions of procedures in a G-path. The controller mechanism uses P, V operations on counting semaphores and a representation of the forwards and backwards incidence matrices of the abbreviated simulating net of the G-path.

A set of variables is associated with the controller of each G-path recording the current marking of the simulating net and the two incidence matrices. The controller contains the following declarations:-

```
bit array (1::no_places, 1::no_transitions) forwards, backwards;
semaphore array  (1::no_procedures_in_G-path) S;
bit array (1::no_places) mark;
semaphore mutex;
```

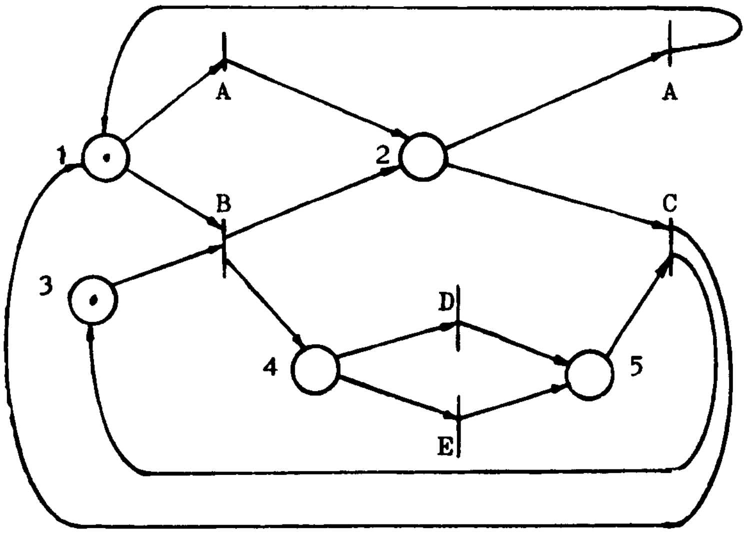
The semaphore mutex is used in the controller to provide a critical section in which the variables may be accessed and altered. Mutex is initialized to one. The semaphores used to synchronize the procedures of the G-paths are declared as an array S whose dimension corresponds to the number of procedures. The semaphores are initialized to zero. The forwards and backwards incidence matrices of the abbreviated simulating net of the G-path are held in two arrays of bits. These two arrays are initialized with the values of the corresponding matrices. Since several transitions in the abbreviated simulating net may have the same name, the matrices may have several columns representing the synchronization of the same procedure.

For example, the G-path:-

<u>path</u> ((A ,B) ; (A , C))* <u>end</u>

<u>path</u> (B ; (D,E) ; C)* <u>end</u>

has the following abbreviated simulating net:-



(The places are labelled with numbers so that they may be identified with the rows of the incidence matrices below.)

The incidence matrices corresponding to this net are:-

place	forwards matrix transitions						backwards matrix transitions						initial marking (mark)
	A	A	B	C	D	E	A	A	B	C	D	E	
1	1	0	1	0	0	0	0	1	0	1	0	0	1
2	0	1	0	1	0	0	1	0	1	0	0	0	0
3	0	0	1	0	0	0	0	0	0	1	0	0	1
4	0	0	0	0	1	1	0	0	1	0	0	0	0
5	0	0	0	1	0	0	0	0	0	0	1	1	0

The incidence matrices have two columns corresponding to the procedure A. The array of bits called mark is initialized with the initial marking and will be used to hold the current marking of the simulating net.

Each synchronized procedure has a prologue and an epilogue which corresponds to the controller. The prologues and epilogues are shown below.

Each procedure has a local constant 'semaphore_of_procedure' which is the index into the array of semaphores S of the semaphore which will be used to delay processes executing that procedure until the synchronization specification permits the procedure to be executed. The array of constants called 'column' are indices to the columns in the forwards and backwards arrays which correspond to that procedure. Where an array is indexed by* this is meant to imply that a whole column from a two dimensional array is to be used. The operations complement, and, xor and or are logical operations on vectors of bits.

The prologue of a procedure is as follows:-

```
 begin  (comment start of prologue)
      constant index semaphore_of_procedure = k;
      constant index array (1 :: number_of_columns_for_this_procedure)
                                column = (v1, v2, ..., vr);
      index j;
      index n;
      logical found;
start:found:=false;  n:=1;
      (comment for each element in column)
```

```
while ¬ found and n ≤ r do
if  complement(mark and forwards(*,column(n)) ) = 0
   then found:=true else n:=n + 1;

if found then begin (comment change mark)
                  mark:=mark xor forwards(*,column(n));
              end
         else
         begin (comment wait until procedure may be executed)
              V(mutex);
              P( S(semaphore_of_procedure) );
              (comment on awakening, calculate new mark)
              go to start;
         end;

j:= schedule_from(mark,forwards,S);
if j = 0 then
         begin (comment no further processes to awaken)
              V(mutex)
         end
         else
         begin (comment awaken process and leave it in a critical
                section)
              V( S(j) );
         end;
(comment rest of prologue and procedure body; n still
    has the value of the index of the element of column
    which was used to alter mark )
```

The procedure 'schedule_from' chooses a value i which is an index into the semaphore array S such that:-

$S(i) < 0$ (that is, there are processes waiting to complete P operations on that semaphore)

and

(complement(mark) <u>and</u> forwards(* , m)) = 0

where m is the index in the forwards matrix of a column corresponding to the procedure which is associated with semaphore S(i).

Thus, the value i chosen corresponds to a procedure for which a process is waiting and which the present marking of the simulating net permits to be executed. There may be several values of i possible, and the one chosen will be dictated by a scheduling policy. If no i can be found to satisfy the relationship, a value of 0 is returned.

The epilogue of a synchronized procedure is as follows:-

```
        (comment epilogue of procedure)
        P(mutex);
        mark:= mark or backwards(*,column(n));
        j:= schedule_from (mark,forwards,S);
        if j=0 then
           begin (comment no process to be awakened after the
                 completion of this procedure)
                 V(mutex)
           end
        else
           begin (comment awaken process and leave it in a
                 critical section)
           V( S(j) );
          end
          (comment rest of epilogue)
    end of procedure;
```

The epilogue updates mark from the backwards matrix and invokes 'schedule_from'. The process which is awakened by the V operation will test, at the end of the prologue that it is executing, whether further processes are allowed to continue as a result of the new value of mark. This feature of the controller is required in order to permit the parallelism which G-paths allow (see the example below).

Example of the implementation.

The G-path given earlier will be used to illustrate the implementation mechanism. The following set of procedure executions may occur:-

Initially, mark has the value:- (1 0 1 0 0).

Suppose that the prologue of procedure B is executed by a process.

The value of the complement of mark is:- (0 1 0 1 1).

The value of forwards(*,column(n)) for that procedure when n=1 is:- (1 0 1 0 0).

Thus, (0 1 0 1 1) and (1 0 1 0 0) = 0, and found will be set to true. Since found is true, mark will be altered to the value:-

(1 0 1 0 0) xor(1 0 1 0 0)

which is:- (0 0 0 0 0).

The invocation of 'schedule_from' will return a value zero and the process will leave the prologue, executing a V(mutex).

Suppose a process now executes the prologue of procedure A, the complement of mark is: (1 1 1 1 1).

The value of forwards(*,column(n)) for n=1 is:-

(1 0 0 0 0).

and for n=2:- (0 1 0 0 0).

Thus, both (1 1 1 1 1) and (1 0 0 0 0)
and (1 1 1 1 1) and (0 1 0 0 0) are non-zero and found will be false. Hence the process will execute V(mutex) and suspend itself by

P(S(semaphore_index_of_procedure_A)).

Suppose another process executes the prologue of procedure D. Then, (1 1 1 1 1) and (0 0 0 1 0) is non-zero and the process will suspend itself on P(S(semaphore_index_of_procedure_D)).

Finally, suppose the process executing procedure B begins to execute the epilogue. The value of backwards(*,column(n)) for n=1 is:-

(0 1 0 1 0),

and hence mark is given the value:-

(0 0 0 0 0) or (0 1 0 1 0)

which is:- (0 1 0 1 0).

The procedure 'schedule_from' may choose one of the two procedures A and D to execute next. Suppose that it selects procedure A. Then the epilogue will execute a V(S(semaphore_index_of_procedure_A)) and the process will finish executing the epilogue of procedure B.

The process which suspended itself in the prologue of A will commence executing the prologue again. The complement of mark is:- (1 0 1 0 1),

and the value of forwards(*,column(n)) for n=1 is:-

(1 0 0 0 0).

Thus, (1 0 1 0 1) and (1 0 0 0 0) is non-zero. However, forwards(*,column(n)), for n=2, has the value:- (0 1 0 0 0) and (1 0 1 0 1) and (0 1 0 0 0) is zero. Thus, found will be set to true and mark will be updated to:-

(0 1 0 1 0) xor (0 1 0 0 0)

which is:- (0 0 0 1 0).

The process executing the prologue of A will now invoke the procedure 'schedule_from'. This will select the procedure D and return the value of the index of the semaphore corresponding to the procedure D. The process will execute a V(S(semaphore_index_of_procedure_D)) and will continue executing the procedure A.

The process which suspended itself in the prologue of D will commence executing the prologue again. The complement of mark is:- (1 1 1 0 1),

and the value of forwards(*,column(n)) for n=1 is:-

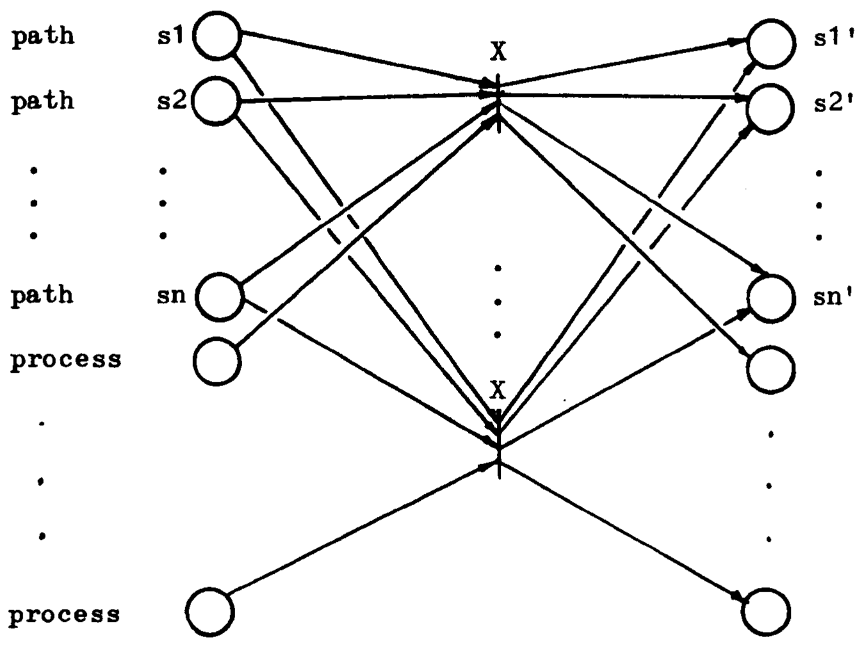
(0 0 0 1 0).

Thus, the value of (1 1 1 0 1) <u>and</u> (0 0 0 1 0) is zero and the process will update mark to be:- (0 0 0 0 0). This time 'schedule_from' will return the value zero, the process will execute a V(mutex) and will begin to execute the body of the procedure D, perhaps simultaneously with the other process executing the body of the procedure A.

Implementation of GE-paths using P, V multiple operations.

The implementation of G-paths can be simplified if the G-paths are GE-paths; that is, composed of E-paths only. One particular implementation for GE-paths uses the synchronization primitives described by Patil (Patil, 1971).

Let a GE-path and a set of processes contain the procedure X. Applying TR6.2 to the path and the processes will give the following abbreviated subnet containing the transitions labelled X:-

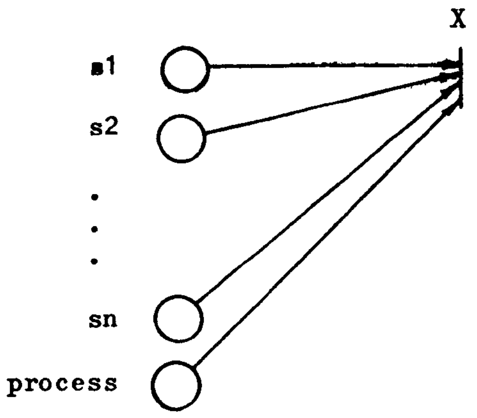


In the above diagrams, places in the paths have been labelled s1,s2,...,sn,s1',s2',...,sn'. Each transition X has one input place from a process and n input places from the GE-path.

We shall define the P multiple operation which is written:-

Pm(s1,s2,...,sn)

as the subnet:-



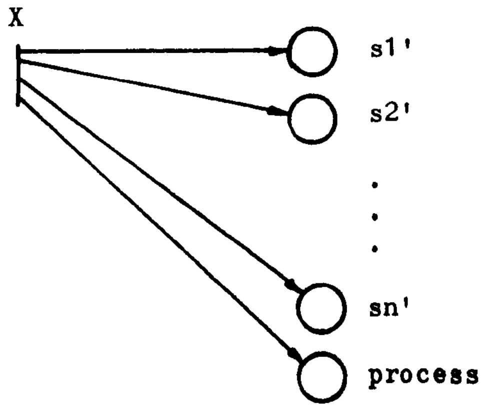
When the P multiple operation occurs it removes a token from each of the places labelled s1,s2,...,sn.

Each place in the GE-path corresponds to a 'semaphore', and the operation 'decrements' each of the 'semaphores' s1,s2,...,sn simultaneously. If any of s1,s2,...,sn are zero (that is, they have no tokens on them in the diagram above), then the Pm operation does not occur immediately but waits until it may proceed. In the diagram above, other transitions may remove tokens from s1,s2,..., sn while the Pm operation is waiting to occur. Hence, the P multiple operation does not prevent other P multiple operations occurring and decrementing any 'semaphore' upon which it is waiting.

The <u>V multiple operation</u>, written:-

Vm(s1',s2',...,sn'),

is defined as the subnet:-



When the V multiple operation occurs, it places tokens on each of the places s1',s2',...,sn'. In terms of 'semaphores', the V multiple operation increments each of the 'semaphores' s1',s2',...,sn' simultaneously.

The implementation of the GE-paths uses a P multiple operation in the prologue and a V multiple operation in the epilogue of the procedures being synchronized. For example, the implementation could be written for a procedure as:-

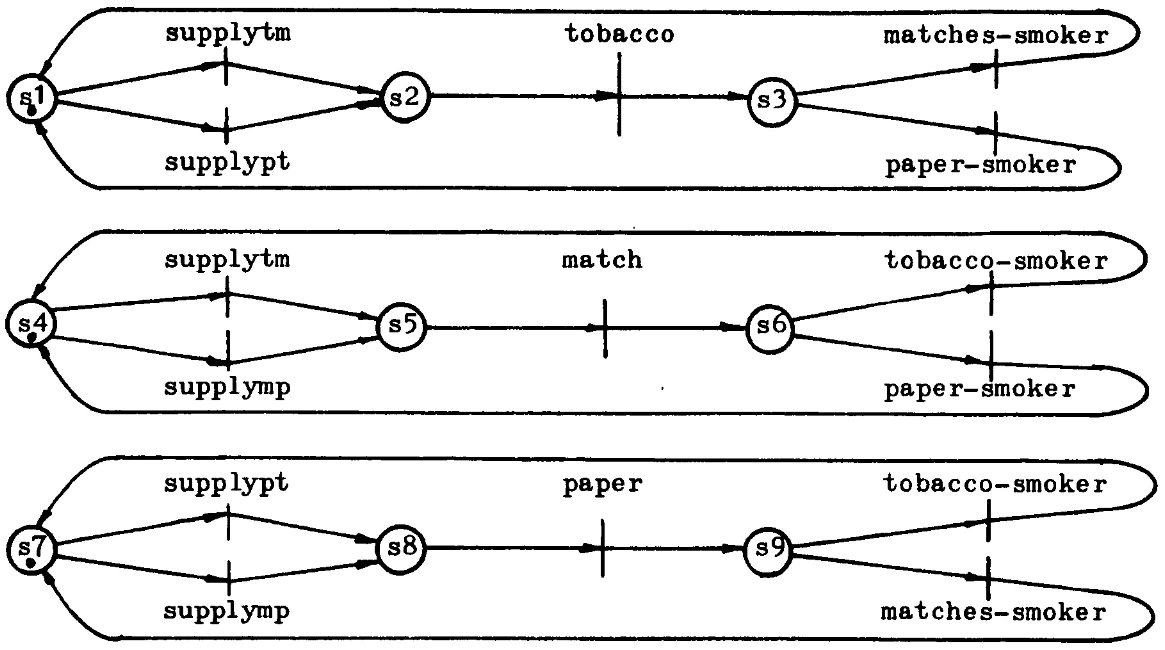
```
procedure X:
    begin
    Pm(s1,s2,...,sn);
    (comment, rest of prologue)

    (comment, body of procedure X)

    (comment, start of epilogue)
    Vm(s1',s2',...,sn');
    end;
```

The 'semaphores' and operations may be generated from the State Machine simulating nets of the individual E-paths composing the GE-path in the following way:-

1) Represent each place in the simulating nets by a unique 'semaphore' initialized to zero if the place is empty, or initialized to one if the place contains a token.

2) For each given procedure appearing in the GE-path generate:-
(i) a P multiple operation with, as arguments, the 'semaphores' representing all the input places to the transitions labelled by the name of that procedure,
(ii) a V multiple operation with, as arguments, the 'semaphores' representing all the output places to the transitions labelled by the name of that procedure.

For example, the individual state machine simulating nets of the GE-path expression given for the cigarette smokers problem earlier in this chapter are:-



The places above have been labelled with the names of unique 'semaphores'. From the net representation we would generate the following implementation:-

```
semaphore s1,s2,s3,s4,s5,s6,s7,s8,s9;
s1:=s4:=s7:=1;    s2:=s3:=s5:=s6:=s8:=s9:=0;
procedures
supplytm:       begin Pm(s1,s4);  (body of supplytm); Vm(s2,s5) end,
supplypt:       begin Pm(s1,s7);  (body of supplypt); Vm(s2,s8) end,
supplypm:       begin Pm(s4,s7);  (body of supplymp); Vm(s5,s8) end,
match:          begin Pm(s5);     (body of match);    Vm(s6)    end,
tobacco:        begin Pm(s2);     (body of tobacco);  Vm(s3)    end,
paper:          begin Pm(s8);     (body of paper)  ;  Vm(s9)    end,
matches-smoker:begin Pm(s3,s9);   (body of m-smoker); Vm(s1,s7) end,
paper-smoker:   begin Pm(s3,s6);  (body of p-smoker); Vm(s1,s4) end,
tobacco-smoker:begin Pm(s6,s9);   (body of t-smoker); Vm(s4,s7) end.
```

Implementation of P and V Multiple Operations.

A simple mechanization of P,V multiple operations can be made using a list structure, a critical section, P and V operations, and a scheduling scheme. The list structure associates with each P,V multiple operation 'semaphore' a queue of items representing requests for Pm operations to be performed on that 'semaphore'. The suspended Pm operation on a set of 'semaphores' is represented by a Pm operation request in the queue of each of these 'semaphores'. Processes are actually suspended using their own private semaphore and the location of these are also included in the items on the queues for the 'semaphores'. The critical section is used to synchronize modifications to the list structure. A scheduling decision must be taken when a Vm operation increments a 'semaphore' upon which processes are waiting.

We shall not describe further the details of this implementation because we have already shown that an implementation exists for G-paths.

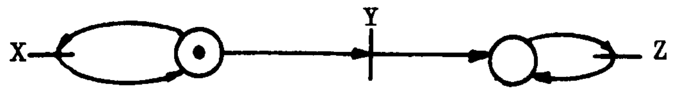
6.6 A DESCRIPTION OF PETRI NETS BY G-PATHS.

GE-paths, although only a subset of the G-path notation, are nevertheless a very convenient and powerful method of describing synchronization. To illustrate this, we will show that GE-paths can be used to describe a class of General Petri nets. We will use the following definition to help us describe this class:-

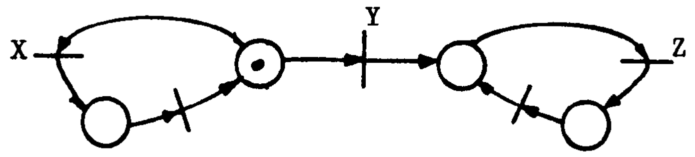
A Petri net is <u>well-connected</u> if and only if:-

1) Every transition has at least one input and one output place (2.6).
2) Every place is an output, but not an input, place of at least one transition.
3) Every place is an input, but not an output, place of at least one transition.

For example, the following Petri net is not well-connected:-



In the above net, the marked place is an output place of one transition, that which is labelled X. However, it is also an input place of X, and hence the net is not well-connected. Similarly, the unmarked place is an input place solely of the transition Z, but it is also an output place of that transition. However, a very similar net does exist which is well-connected:-



<u>Remarks</u>: The first net could be regarded as an abbreviation of the second well-connected net (see the abbreviation 3a). Thus, although the subclass of well-connected nets may seem restrictive, a proportion of the excluded

General Petri nets may be manipulated into the required form.

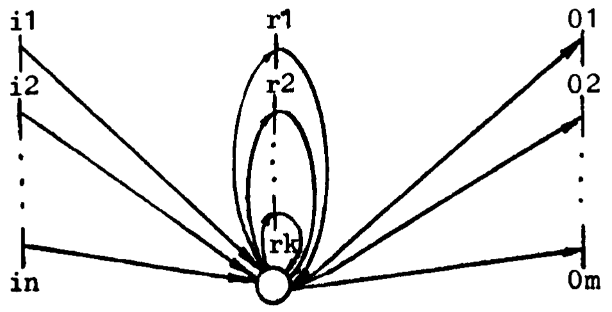
Theorem 6.3: A safe, well-connected Petri net can be described by a GE-path whose simulating net will generate exactly the set of actions generated by that Petri net.

Proof: For the proof we shall describe a transformation which will generate a GE-path from any given safe, well-connected Petri net. We will then show that the simulating net for such a GE-path generates exactly the set of actions generated by the safe, well-connected net.

The transformation is as follows:-

Label each transition of the Petri net with a unique name.

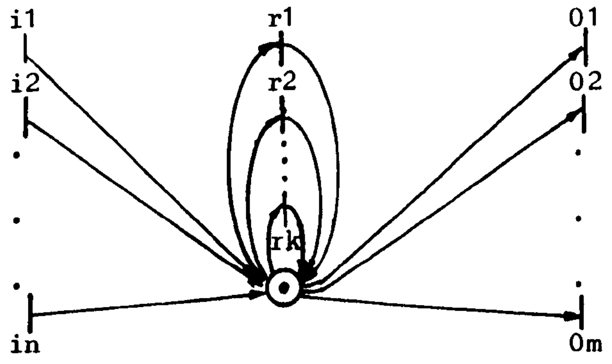
For each unmarked place in the net:-



include the following E-path as a component of the GE-path:-

path ((i1,i2,...,in) ; (r1,r2,...,rk)* ; (01,02,...,0m))***end**

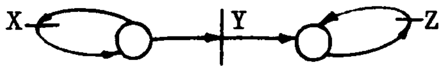
For each marked place in the net:-



include the path:-

path ((r1,r2,...,rk)* ; (01,02,...,0m) ; (i1,i ,...,in))*end

(Note that once the place is marked, then the initial transitions (2.6) of that place cannot fire and put another marker on that place without the terminal transitions having fired in this path expression description of the net. Hence, the transformation only represents safe nets. In addition, the transformation assumes that the net is well-connected. We shall ignore the case of transitions which do not have an input place or an output place. Consider the earlier example of a net which is not well-connected:-



If we apply the transformation to it we obtain:-

path ((X,Y) ; X)*end

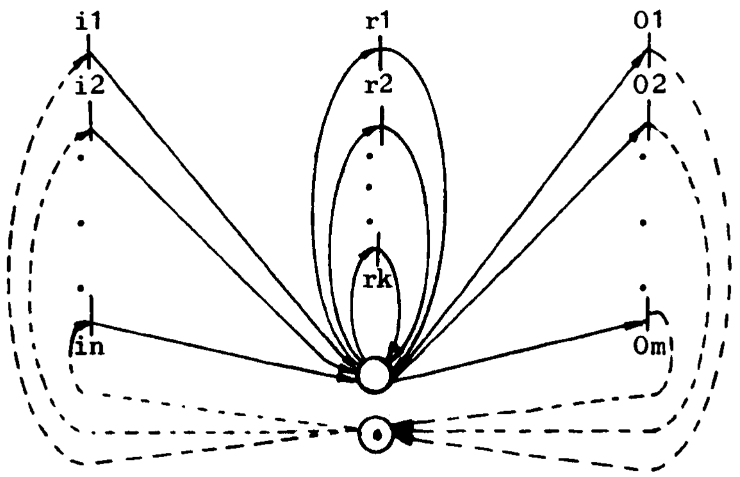
path ((Y,Z) ; Z)*end

However, this GE-path does not model the net above since the first path permits further 'X's to occur after the occurrence of a 'Y', and the second path allows 'Z's to occur before a 'Y' occurs.)

We shall next show that the component E-paths of the GE-path yield an equivalent net to the original net when translated into a simulating net. The path:-

path ((i1,i2,...,in) ; (r1,r2,...,rk)* ; (01,02,...,0m))*end

has the simulating net:-

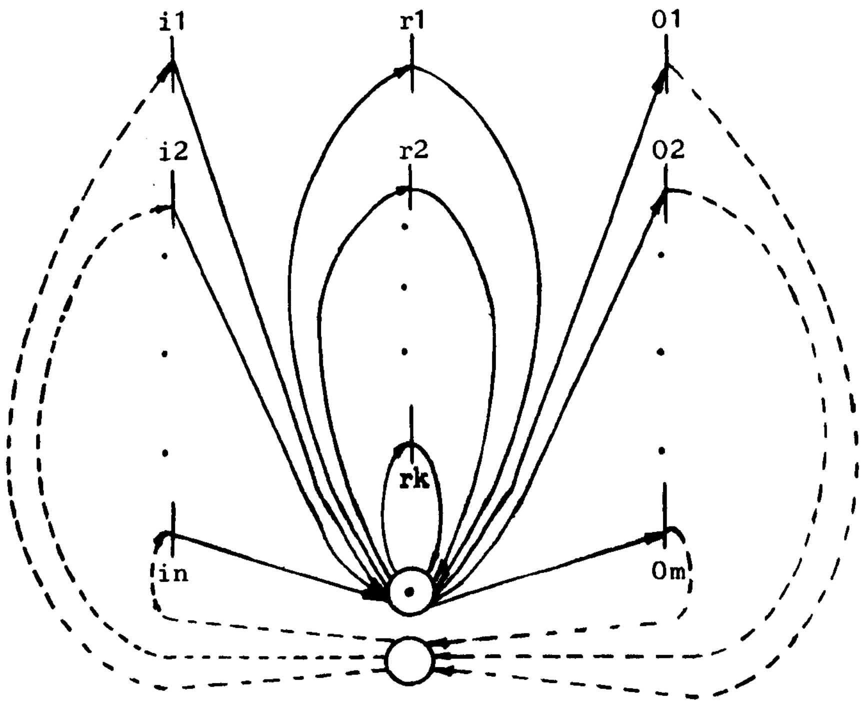


The unmarked place and transitions are all identical to those of the original net. However, we have introduced a new marked place and additional arcs (the additional arcs are dashed in the diagram above). The place and dashed arcs ensure that the unmarked place is safe (the original net was safe) but do not affect in any way the order in which the transitions i1,i2,...,in or 01,02,...0m, or r1,r2,...,rk may occur.

The path:-

<u>path</u> ((r1,r2,...,rk)* ; (01,02,...,0m) ; (i1,i2,...,in))* <u>end</u>

has the simulating net:-



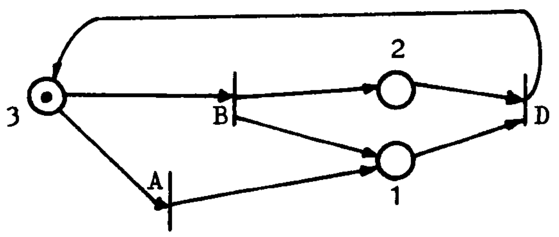
This time an unmarked place is introduced. The extra arcs are again distinguished by dashes. Using the same arguments as before, we conclude that this net allows the same transitions to occur in a similar order to the original net.

In generating the GE-paths from the original net we only use the places and the initial and terminal transitions to those places.

A given transition may have arcs to several places and this is accounted for by the occurrence of the name of that transition in as many E-path expressions within the GE-path expression. The intersection transformation TR6.1 guarantees that on generating a simulating net for a GE-path, the names of the transitions will be inspected, and those which are identical used to recreate the original transition.

Q.E.D.

Example: The safe, well-connected net:-



gives a GE-path:-

path ((B,A);D)* end (for the place labelled 1)

path (B ; D)* end (for the place labelled 2)

path ((B,A);D)* end (for the place labelled 3).

The net is not live (2.4) because the firing of transition A will prevent any other transition from firing. This is reflected in the GE-path expression where one can see that the second path expression only permits the procedure execution of B and D to alternate, and the first path expression permits a D to be followed by an A and then a further D (excluding B).

6.7 COMPARISONS BETWEEN PATH NOTATIONS.

To complete the discussion of General Path expressions we shall compare them with the previous path notations.

GE-paths and R-paths.

We may express the R-path:-

path (A ; B; A ; C)* end

as the following GE-path:-

path (A ; (B , C))* end

path (B ; C)* end.

However, there are R-paths for which a direct translation into an equivalent GE-path is not possible. For example:-

path (A ; A ; B)*end

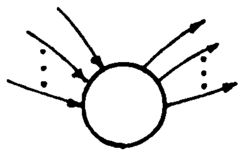
We will show that this corresponds to a difference in declarative power between R-paths and GE-paths rather than any difference in scheduling power (4.6).

From Theorem 6.3, any safe, well-connected Petri net can be expressed as a GE-path. Thus, the synchronization of any R-path which has a simulating net which is well-connected (all R-paths have simulating nets which are safe by Theorem 5.3) may be expressed by a GE-path.

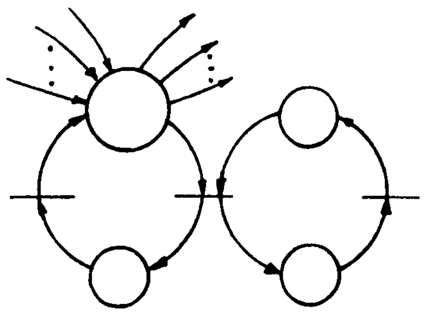
Lemma 6.4: Any simulating net of an R-path may be transformed into an equivalent well-connected simulating net.

Proof: By construction, every transition of the simulating net will have at least one input and one output place. Examine each place in turn. If every place satisfies the conditions 2) and 3) for a place in a well-connected Petri net, then the net is well-connected. However, if the net contains places which do not satisfy conditions 2) and 3), then we can transform them into the appropriate form by adding arcs, places and transitions which are not live.

Thus, suppose we have a place:-



We make the following transformation:-



The place resulting from the transformation is of the required form, being an output (but not an input) place of one transition and an input (but not an output) place of another. Since these transitions are not live, they will not alter the behaviour of the net. Hence, the simulating net of an R-path may be transformed into a well-connected simulating net.

Q.E.D.

Theorem 6.5: A GE-path may be written to express the synchronization of any R-path.

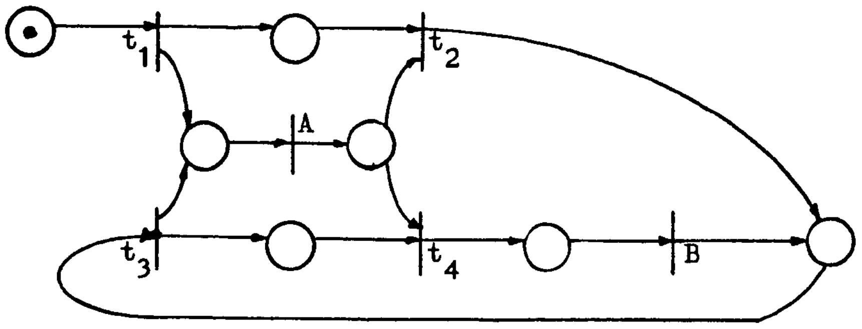
Proof: From Lemma 6.4, the simulating net of an R-path may be transformed into an equivalent well-connected simulating net. However, by Theorem 5.3 the simulating net of an R-path is safe. Hence, by Theorem 6.3, we may construct a GE-path which expresses the synchronization given by the safe, well-connected simulating net of the R-path.

Q.E.D.

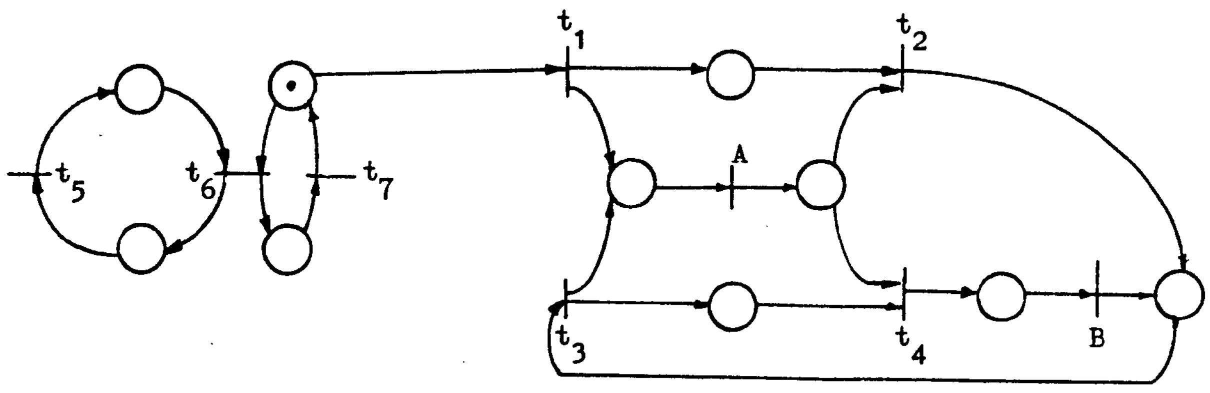
Example: The R-path:-

path A ; (A ; B)*end;

has the simulating net:-



which may be transformed into the equivalent well-connected simulating net:-



and may be written as the GE-path:-

path $(t_1 ; t_2)^*$ end path $(t_3 ; t_4)^*$ end

path $((t_1,t_3) ; A)^*$ end path $(A ; (t_2,t_4))^*$ end

path $(t_4 ; B)^*$ end path $((B,t_2) ; t_3)^*$ end

path $((t_1,t_6) ; t_7)^*$ end path $(t_6 ; t_7)^*$ end

path $(t_6 ; t_5)^*$ end path $(t_5 ; t_6)^*$ end

The GE-path written to express the synchronization of an R-path does not include any additional scheduling policy, and hence we conclude that the scheduling power of synchronization of GE-paths is at least that of R-paths. However, in the above example we have to include, in the GE-path, additional procedure names and these would require further processes to invoke them. Hence, the declarative power of GE-paths is insufficient to describe the synchronization given in an R-path. To prove equivalence between the scheduling power of GE-paths and R-paths it is sufficient to remark that in 6.5 we gave an implementation for G-paths (which includes GE-paths as a subset) in terms of R-paths and that the implementation did not involve any additional scheduling and all conflicts of G-paths were expressed as conflicts of R-paths. That particular implementation introduced additional procedures in order to express the synchronization. In general, the declarative power of R-paths is insufficient to describe the synchronization given in a GE-path. For example, R-paths cannot express the possible simultaneous execution of two different procedures by processes as in the GE-path below without the use of some additional programming constructions:-

```
path ( A ; B )* end
path ( A ; C )* end.
```

(Procedure B and C may be executed simultaneously by two different processes.)

G-paths and R-paths.

Since R-paths are contained in the G-path notation we conclude that the scheduling and declarative power of GR-paths is at least that of R-paths. From 6.5, G-paths may be implemented using R-paths and hence we conclude that G-paths and R-paths are equivalent in terms of scheduling power. However, R-paths have insufficient declarative power to describe GE-paths and hence we conclude that G-paths have greater declarative power than R-paths.

GE-paths, G-paths and E-paths.

R-paths have greater scheduling power than E-paths (see Chapter 5) and we have seen that R-paths are equivalent to GE-paths and G-paths with respect to scheduling power. Therefore we conclude that the scheduling power of GE-paths and G-paths is greater than that of E-paths. R-paths have greater declarative power than E-paths, but R-paths have insufficient declarative power to describe GE-paths or G-paths. Since E-paths are contained in both the GE-path and G-path notation, we conclude that GE-paths and G-paths have greater declarative power than E-paths.

GE-paths and G-paths.

Since both GE-paths and G-paths are equivalent in scheduling power to R-paths, we conclude that G-paths are equivalent in scheduling power to GE-paths. GE-paths are contained in the G-path notation, and GE-paths have insufficient declarative power to describe R-paths which are also contained in the G-path notation. Hence we conclude that G-paths have greater declarative power than GE-paths.

6.8 SUMMARY OF GENERAL PATH EXPRESSIONS.

General path expressions allow the synchronization and co-ordination of a given action to be specified simultaneously by separate synchronization expressions. In particular, they permit descriptions of synchronization which involve the possible simultaneous or concurrent execution of different procedures by processes and have a greater declarative power than R-paths and E-paths. We have given several implementations of G-paths including one using R-paths.

The synchronization specified by a class of safe General

Petri nets can be described by means of a set of GE-paths which are G-paths composed of Elementary paths. This would suggest that the General path notation has sufficient scheduling power to describe many synchronization problems. The equivalence between R-paths and G-paths in terms of scheduling power implies that this result also applies to R-paths.

CHAPTER 7

ALTERNATIVE PATH NOTATIONS.

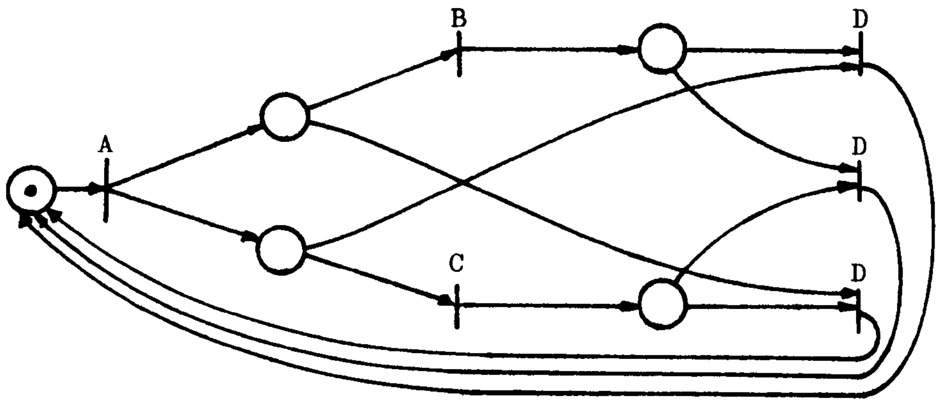
The path notations that were discussed in earlier chapters were constructed using a regular expression or set of regular expressions as a basis for specifying the synchronization. The notations include sequence, selection and repetition and have the characteristic that each action described in any individual expression can only be executed by mutually exclusive processes. The remarks which follow investigate other approaches to the specification of synchronization by path expressions, for example, the use of variables within a path expression and a different interpretation of sequence, selection, and repetition which permits parallel executions by processes of actions within an individual path. Wherever possible, we shall outline the problems and benefits of these alternatives and compare them with the earlier notations.

7.1 THE INCLUSIVE OR CONSTRUCTION.

The first construction we shall consider is the inclusive or, represented by $\oplus$. Informally, we define $A \oplus B$ to mean that either the procedure A or the procedure B may be executed by a process or both may be concurrently executed by different processes. The inclusive or could be used in either E-path or R-path expressions to increase their declarative power of synchronization. However, such E-paths or R-paths cannot be directly represented as a state machine because of the possibility of the simultaneous execution of procedures by processes. The generation of a simulating net for the construction is difficult and is complicated by special cases. For example, the path:-

<u>path</u> (A ; (B $\oplus$ C) ; D)* <u>end</u>

could be represented by the simulating net:-



(Note the abbreviation for procedure calls to D in the above net.)

If the inclusive or is used within a selection, the simulating net becomes very complex to construct. For example, a simulating net of the path:-

<u>path</u> (A ; (B $\oplus$ C) , (D $\oplus$ E $\oplus$ F) ; G)* <u>end</u>

involves ten transitions representing the calling mechanism for G if the simulating net is constructed in the same manner as the one above. Further research is required to find a convenient mechanism with which to define the inclusive or construction.

Implementation of inclusive or involves similar problems to those described above for its definition. Using a similar scheme to the one proposed in Chapter 6 to implement G-paths in terms of R-paths, it is possible to describe an implementation for the $\oplus$ construction. As an illustration of this implementation, we give the R-path and corresponding procedures which would be required to implement the example above:-

```
path ( A ; ( (start_B ;
                ( (finish_B ;
                     ( (start_C ; finish_C ; D) , D) )  ,
                 (start_C ;
                     ( (finish_B ; finish_C),
                       (finish_C ; finish_B)) ; D  ) ) ) ,
              (start_C ;
                ( (finish_C ;
                     ( (start_B ; finish_B ; D) , D) ) ,
                 (start_B ;
                     ( (finish_C ; finish_B) ,
                       (finish_B ; finish_C) ); D   ) ) ) )   )* end;
```

```
procedure A :  begin (body of A) end;
procedure B :  begin start _B ; (body of B) ; finish_B end;
procedure C :  begin start_C ;  (body of C) ; finish_C end;
procedure D :  begin (body of D) end;
```

where start_B, start_A, finish_A, finish_A are declared as procedures with null bodies.

However, such implementations of path expressions using the inclusive or may generate very large R-paths if they involve the possible simultaneous execution of many procedures by processes. Finally, since paths involving the construction may be implemented using R-paths, we conclude that such paths have the same scheduling power as R-paths, and hence, by the results of Chapter 6, as G-paths.

7.2 THE SIMULTANEOUS EXECUTION CONSTRUCTION.

The simultaneous execution construction was introduced in the original path notation of Chapter 1 and by Campbell and Habermann (Campbell and Habermann, 1974) as a useful programming tool. It provides the following facilities (see examples in Campbell and Habermann):-

1) A means for specifying simultaneous execution of a given procedure by several processes.

2) A counting facility which can be used in the synchronization expression.

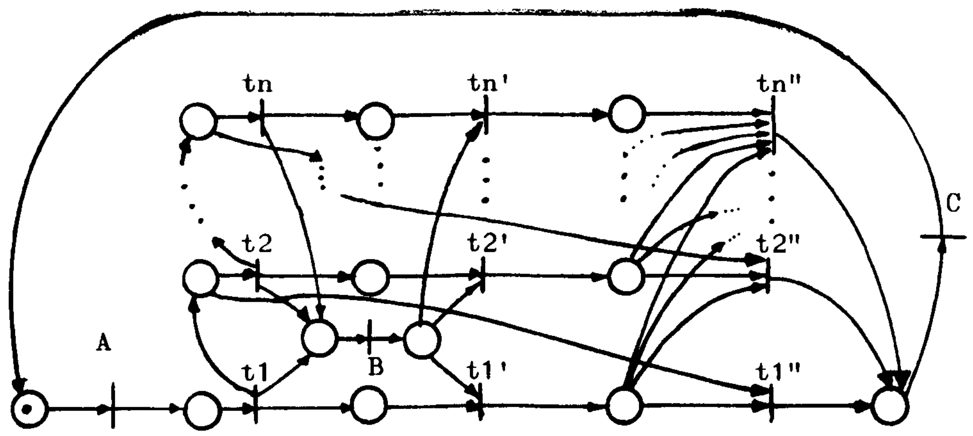
3) A limited ability to program priority within Path expressions.

Simultaneous execution is denoted by a bracket pair { } around an expression and allows as many instances of that expression to become available to be executed as there are requests for it up until the moment when all the instances which have been requested have been completed (See chapter 1).

Simulating nets may be generated for E-paths involving the simultaneous execution construction. For example, the path expression:-

<u>path</u> (A ; { B } ; C)* <u>end</u>

has the simulating net:-



The transitions labelled t1,t2,..tn and t1',t2',...,tn' represent the flow of synchronization control for invocations of the expression within the simultaneous execution. They are similar to the control mechanism of the R-paths for repeated procedure names and also keep count of the number of concurrent executions of the procedure B. We assume that sufficient of these transitions (that is,n) are provided to meet the requirements of the program in which the path is used. The occurrence of one of the transitions t1",t2",...;tn" represents the condition occurring that all the instances of the executions of B which have been requested have now been completed. Lauer (Lauer, 1974) has also given a simulating net for the construction, but both his net and the one given above are unavoidably clumsy because of the necessity of describing a counting mechanism in terms of a Petri net.

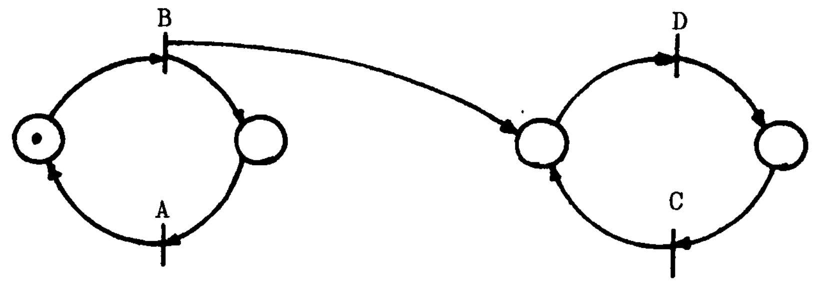
The implementation for simultaneous execution given by Campbell and Habermann (Campbell and Habermann, 1974) may be easily adapted to permit the implementation of E-paths and simultaneous execution using counting semaphores and counters. Implementation of simultaneous execution in GE-paths is more difficult, however there are no major problems to such extensions. For example, the controller described in Chapter 6 for GE-paths may be extended to implement simultaneous execution by including counters and further state information. However, the implementation given using P,V multiple operations cannot be extended in this way. Implementation of simultaneous execution in R-paths introduces problems of representing non-determinancy. Consider the following R-path:-

<u>path</u> ({ A ; B ; A } ;D)* <u>end</u>

If the procedures A and B have been executed by processes and another process executes an A, then we require some mechanism to permit either a B or a D to be executed. Implementations for simultaneous executions used within R-paths require further research.

The use of the simultaneous execution with GE-paths allows certain well-connected General Petri nets which are not safe to be described. For example:-

The General Petri net:-



can be described by the GE-path:-

```
path (B ; A)*end;
path ( { (B,C) ; D } )*end;
path ( {D ; C}  )*end;
```

The declarative power of the E-path notation when supplemented with simultaneous execution is greatly increased because we can write synchronization expressions which involve parallelism between executions of the same action, and which involve primitive counting facilities as, for example, when used to describe P,V operations on counting semaphores:-

A type declaration for a counting semaphore:

```
type  semaphore;
      path { V ; P } end;
operations P,V
endtype;
```

The type semaphore above may be used to declare instances of semaphores initialized to zero. For semaphores initialized to other values we should require additional notation to state the initial status of the path expression. The bodies of the P,V operations are empty.

7.3 THE PARALLEL CONSTRUCTION.

Synchronization specifications permitting simultaneous execution of procedures by different processes can be programmed using General Path Expressions using the inclusive or construction, or using the simultaneous execution construction. Another, more direct, mechanism to specify the possible parallel execution of actions is to use a parallel construction which is analogous to the parbegin, parend construction found in Algol68 (Algol 68, 1968). We shall denote the parallel construction by '||'.

The expression:-

A || B

specifies that the procedures A and B may both be executed, possibly simultaneously, by one process.

The expression:-

(A || B) ; C

specifies that a procedure C cannot be executed by a process until the procedures A and B have been executed by processes. The path expression:-

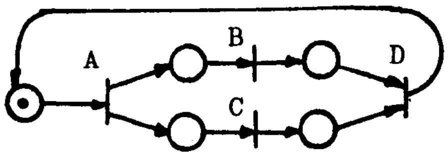
<u>path</u> (A ; (B || C) ; D)* <u>end</u>

specifies the same synchronization as the G-path:-

<u>path</u> (A ; B ; D)* <u>end</u>

<u>path</u> (A ; C ; D)* <u>end</u>

and could be represented by the simulating net:-



Further research is required to provide a formal definition for the parallel construction when it is used within E-paths, R-paths and G-paths.

<u>E-paths and a limited Parallel Construction</u>.

Because E-path expressions have less declarative power but are simpler to implement than G-paths and R-paths, there is an advantage in considering extensions to this notation which will increase its power without making the implementation more complex. One such possible extension is to include within the E-path notation a Parallel construction which is limited by the definition of the notation to appear in E-path expressions only where a simple implementation is possible. (A similar limitation was imposed upon the repetition construction in Chapter 3.) For example, the path expression:-

```
path ( A ; (B || C) ; D )* end
```

has a simple implementation in the form of:-

```
semaphore s1,s2,s3,s4,s5; s1:=1; s2:=s3:=s4:=s5:=0;
procedures
    A: begin P(s1); (body of A); V(s2); V(s3)end;
    B: begin P(s2); (body of B); V(s4) end;
    C: begin P(s3); (body of C); V(s5) end;
    D: begin P(s5); P(s4) ; (body of D) ; V(s1)end;
```

The parallel construction must be limited in its use in an E-path expression because certain expressions cannot be easily implemented. Consider for example:-

```
path ((( A || B ) , C ) ; D )* end.
```

The implementation will use different semaphores in the P operations of the respective prologues of the procedures A and B. However, the prologue of procedure C must contain a P operation on both of these semaphores, but it cannot decrement them simultaneously, Hence, we have the possibility that a process executing procedure C may decrement one of these semaphores, and that one of the procedures A and B may be executed by a different process which decrements the other semaphore. The result is that only one of the procedures A and B may be executed and none of the procedures in the path expression may be executed by further processes.

A suitable set of productions can easily be devised for the E-path notation and parallel construction which define only those expressions which may be implemented in the manner outlined above.

7.4 OPEN PATH EXPRESSIONS.

The purpose of this section is to give an example of a notation which is different from those we have discussed earlier and is not based on the regular expression. We will demonstrate that this path notation has more declarative power than the E-path notation and has an implementation.

The notation is called an open path notation because, unlike the previous notations, it will have no enclosing critical section. Sequence and selection will differ slightly in meaning from their earlier definition and the notation may specify the absence of synchronization. In Chapter 1 we referred to associating path expressions with type definitions. The open path expression would permit language designers to enforce, as a necessary part of the type definition in some language, a complete description of how every operation on that type is to be synchronized, eliminating the risk of a programmer overlooking some synchronization constraint.

We shall give a brief description of the notation, giving the semantics and implementation together in the form of a transformation from the notation to P, V operations in the prologues and epilogues of procedures. Procedure names are restricted (as in E-paths) to occurring only once in any open path expression.

Example syntax of Open path notation.

```
P1: path ::= path  list end
P2: list ::=       sequence [ , list ]
p3: sequence ::=item [ ; sequence]
p4: item ::= ( list )
           | n:( list )
           |   {list}
           | procedure name.
```

For simplicity we shall not allow nesting of simultaneous execution (again denoted { }).

Implementation of Open path expressions.

An open path expression can be implemented using P and V operations on counting semaphores in the prologues and epilogues of the procedures which are named by that path expression. The following recursive algorithm will translate open paths into this implementation. At each stage of the algorithm the path expression yet to be translated is labelled by the syntactic entity that it represents in the production rules above. In general, the path expression to be translated will be surrounded by two generated synchronization operations O_L and O_R which are on its left and right respectively. The operation O_L may be null, a P or a PP operation. (To simplify the algorithm two operations PP and VV are introduced which take three parameters, a counter and two semaphores. These operations will be explained later in terms of P and V.) The operation O_R may be null, a V or a VV operation.

Stage 1) Replace <u>path</u> list <u>end</u> by null list null

(The first production does not introduce any synchronization.).

Stage 2) Apply the following set of transformation rules in the order given by a parse of the path expression using the production rules above.

a) Replace:- 0_L sequence , list 0_R by:-

0_L sequence 0_R

and

0_L list 0_R

(The comma differs slightly from its use in E-paths in that it is used as a distributive mechanism for the synchronization in which it is embedded.)

b) Replace:- 0_L item ; sequence 0_R by:-

0_L item V(s1)

P(s1) sequence 0_R

and declare a counting semaphore s1 with initial value 0.

(The semicolon is used to introduce a sequence in almost the same manner as for E-paths. The semaphore is a counting semaphore, and the operations corresponding to 0_L and 0_R may be null or PP, VV operations as well as P,V operations.)

c) Replace:- 0_L n: (list) 0_R by:-

P(s2) 0_L list 0_R V(s2)

and declare a semaphore s2 initialized to n.

(This construction permits up to n simultaneous executions by processes of procedures in list, subject to the restrictions imposed by the synchronization in which it is embedded.)

d) Replace:- 0_L { list } 0_R by:-

$PP(c,s,0_L)$ list $VV(c,s,0_R)$

and declare a semaphore s initialized to one and a counter initialized to zero. (The simultaneous execution construction is exactly that described in 7.2.)

e) Replace:- 0_L procedurename 0_R by:-

```
procedure procedurename: begin
                           0_L;
                           (body of procedure);
                           0_R;
                           end;
```

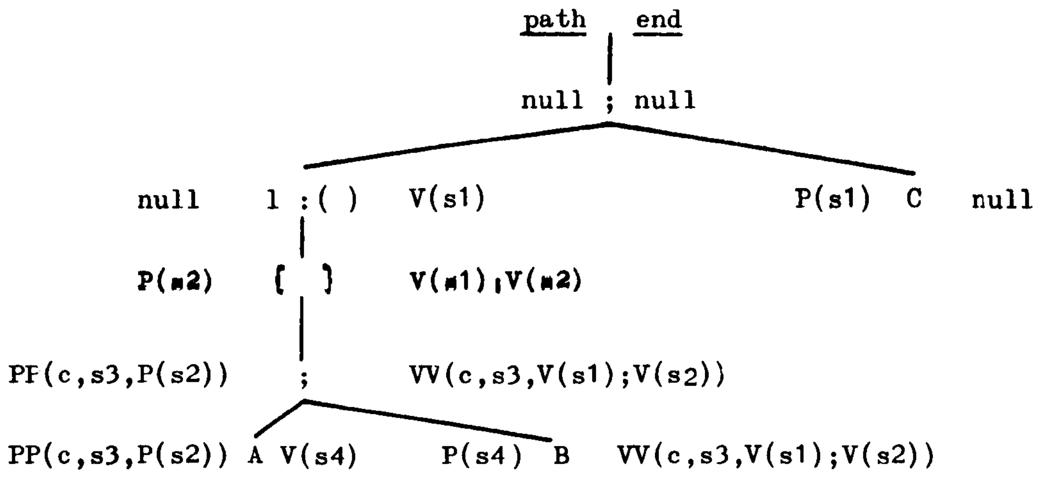
The operation PP and VV implement the simultaneous execution synchronization. Both operations share a counter C1 and a semaphore s. The semaphore s is used to exclude more than one process from changing the counter at a time (Campbell and Habermann, 1974). The PP operation increments the counter, the VV operation decrements it. If the counter is increased from zero the synchronization operation synch is invoked (see below). If the counter is decreased to zero the synchronization operation $synch_1$ is invoked.

```
procedure PP (counter c ; semaphore s; procedure synch):
begin    P(s);
         c: = c+1;
         if c=1 then synch;
         V(s);
end;
```

```
procedure  VV  (counter c ; semaphore  s ,  procedure synch):
begin      P(s);
           c: = c-1;
           if c=0 then synch;
           V(s);
end;
```

The following example illustrates the translation of an open path expression into its implementation. The translation is represented as a tree. Each step of the algorithm corresponds to a node in that tree and the synchronization operations which have been generated up to a given step are written on either side of the corresponding node.

The open path expression:- path 1:({ A ; B }) ; C end translates as follows:-



The resulting set of procedure prologues and epilogues which are created by part e) of the algorithm are written below in program form:-

```
semaphore s1,s2,s3,s4; counter c;
s1:=s4:=0; s2:=s3:=1;   c:=0;
```

```
procedure A: begin PP(c,s3,P(s2)); (body of A); V(s4) end
procedure B: begin P(s4);(body of B); VV(c,s3,(V(s1);V(s2))) end
procedure C: begin P(s1); (body of C); end
```

Execution by a process of procedure A will set semaphore s2 to zero, semaphore s4 to one and the counter to one. Thus, further processes may execute A and one process can execute B. When there are an equal number of executed A and B procedures and no further processes executing A the counter will have been reduced to zero and semaphores s1 and s2 set to one. C may now be executed by a process. As many executions of C are permitted as have been groups of simultaneous executions of procedures A and B.

Examples:

We shall illustrate the open path notation by examples.

The open path:- path 1: (write ; read) end

is equivalent to the E-path:- path (write ; read)*end

The open path notation is equivalent in power to P and V operations. For example, we can include the following open path in a counting semaphore type definition:-

```
path  V ; P end
```

For a binary semaphore we would use:-

```
path  1:( V ; P ) end
```

Some of the open paths which may be written contain redundancies. For example, the open path:-

```
path  8:( A ; 12:( B ) ; C ) end
```

is equivalent to:- path 8:(A ; B ; C) end

The counting facilities inherent in the notation can be utilised in much the same way as the counting facilities of counting semaphores. For example, we pose the following resource allocation problem:-

> Let 10 buffers be shared amongst 3 sets of processes A,B and C. To ensure progress in the 3 sets of processes we ensure that each set of processes has a minimum number of buffers it can always claim from the pool. However, to accommodate heavy demand, further buffers may be allocated to a set of processes which need them, providing that this does not interfere with the basic requirements of the other two sets of processes. The minimum requirements for A,B and C are 2,3 and 1 buffers respectively. One method, which ensures that these minimum requirements are guaranteed, places an upper bound on the use of buffers by each set of processes such that the sum of the upper bounds for any two sets leaves the minimum number of buffers available to the third set. Such a set of maximum bounds for A,B,C are 4,5 and 3, respectively.

The path expressions must synchronize the use of the buffers so that neither are the upper bounds exceeded, nor are there more buffer requests than buffers available. Such a path is:-

```
path  10:( 4:(getbufferA;releasebufferA) ,
           5:(getbufferB;releasebufferB) ,
           3:(getbufferC;releasebufferC))end
```

where getbufferA is a procedure which allocates a buffer to the set of processes A and releasebufferA returns the buffer to the pool. The implementation of such a path for the actions getbufferA and releasebufferA is:-

```
procedures
   getbufferA: begin P(s2);P(s1);(body of getbuffer); V(s3); end,
   releasebufferA: begin P(s3);(body of releasebuffer);V(s1);V(s2) end
```

where s1,s2,s3, are semaphores initialized to 10,4,0 respectively.

Summary of Open paths.

Open paths give an alternative to the path notation we gave earlier and show that such alternatives exist. The notation is as powerful as that of Chapter 1, and can describe counting semaphores in a simple and economical way. In addition the counting facilities may be used as a programming tool as was shown in the last example. The notation illustrates the more general nature of the sequencing and selection constructions and forms a more natural basis in which to embed the simultaneous execution construction since in this notation the "simultaneous execution of expressions" is the general case and critical sections become special cases.

7.5 POSSIBLE MODIFICATIONS TO THE SELECTION AND REPETITION CONSTRUCTIONS.

Within the framework of the path notations of Chapters 3, 5 and 6 we can modify selection and repetition to produce additional ability to describe synchronization. The first such modification is to allow the execution of one procedure, by a process, to be given precedence over the execution of another procedure, by a different process, in a selection. We develop this scheme by introducing into the selection a choice mechanism based on the values of variables. Finally, we apply the same scheme to repetition.

The motivation for this section is the idea of developing a path notation which has the control constructions which are commonly found in conventional programming languages like Algol. This would permit a great flexibility in the programming of synchronization, and we will show that, provided certain restrictions are made, such notations may be practical.

A Precedence Construction.

The precedence construction, denoted by >, may be used in a path expression wherever the comma, used to denote selection, may be used. A > B is a path expression which denotes that there is a selection to be made between the execution of A or B by processes such that if both procedures have been invoked, then the execution of procedure A will have precedence over that of B. Implementations for path expressions containing the precedence construction can be devised.

For example, if the precedence construction is used with E-paths and the E-paths are implemented using the scheme given in 3.5, then the following implementation may be used:-

Consider the path expression:-

```
path ( A > B)* end
```

This may be implemented as:-

```
procedure A: begin P(s);        (body of A);        V(s) end
procedure B: begin P(s1); P(s);(body of B); V(s); V(s1) end
```

where s and s1 are semaphores initialized to 1. Only one process invoking procedure B may compete to execute P(s) against any number of processes requesting A. Assuming that the scheduling mechanism for the processes waiting on semaphore s is a first in, first out queue, precedence will be granted to processes invoking the procedure A. This implementation for the precedence construction may be extended to cover situations where precedence is expressed between three or more procedures, as in:-

```
        path (A > B > C)*end
```

which has for its implementation:-

```
procedure A: begin P(s);(body of A);V(s) end;
procedure B: begin P(s');P(s);(body of B);V(s);V(s'); end;
procedure C: begin P(s");P(s');P(s);(body of C);V(s);V(s');V(s") end;
```

Further research is required to obtain a formalism of the notion of precedence which is applicable to path expressions. This would then allow implementations to be assessed with respect to whether they implement precedence correctly, and permit the measures of absolute and declarative power of synchronization of path expressions to be applied to path notations containing precedence constructions.

Selections with a Boolean control mechanism.

To increase the versatility of the path expression and to make it more analogous to the traditional programming language we could allow the selection construction to include a control mechanism analogous to the if, then statement of Algol (Algol 68, 1968). The construction described below has similarities to Dijkstra's Guarded Commands (Dijkstra, 1975). Each element of a selection may be prefixed by a 'guard' consisting of an expression which may be evaluated, without side effects, to a boolean value. A selection is made by choosing one of the components of the selection whose guard is true and which has been invoked. A component of the selection with no guard is regarded as having a guard which is true. A possible production to define this construction is as follows:-

```
selection ::= boolean expression | element   [, selection]
```

Example:
```
        path ( a > 0 | read,  a < 10 | write )* end
```
describes the synchronization for a buffer which may store 10 messages. The value of a gives the number of messages which have been written but not yet read.

We shall impose the following constraints upon the guards:-

1). The guards of a selection are evaluated by the epilogue of the procedure whose execution leads to the selection taking place. This prevents the continuous evaluation of the guards.

2). The procedures of a Guarded path may only be executed by mutually exclusive processes. Problems would arise if variables in the guards could change during the evaluation of the guards in an epilogue.

3). The variables composing a guard may only be changed within a procedure belonging to the path expression in which they are used as a guard. This ensures that 2) is effective.

Example A:

Consider the problem of allocating buffers to the three sets of processes described in 7.4. The problem may be programmed as:-

```
type allocator;
integer countA,countB,countC;     integer available;
countA:=countB:=countC:=0; available:=10;
path ( available > 0 | (    countA <=4 | getbufferA ,
                            countB <=5 | getbufferB ,
                            countC <=3 | getbufferC )   ,
        countA > 0 | releasebufferA ,
        countB > 0 | releasebufferB ,
        countC > 0 | releasebufferC )*   end;
  procedure getbufferA:  begin  countA:=countA + 1;
                                available:=available -1;
                                . . .
                         end;
  .
  .
  .
  procedure  releasebufferA:  begin
                                  countA:=countA -1;
                                  available:= available + 1;
                                  . . .
  .
  .                           end;
  .
  operations getbufferA,getbufferB,...,releasebufferC
  endtype;
```

This example illustrates guards which are nested. A getbuffer operation can only be performed by a process if the set of processes to which it belongs has not exceeded the maximum number of buffers the set is allowed, and if there are buffers available.

The guards for a given selection must always permit one of the possible choices to execute otherwise the path may result in permanently blocking processes. Implementation could be accomplished by a similar use of tables of incidence matrices or State machines as described in the implementation of R-paths and GE-paths and GR-paths. To be executed, a procedure must be a possible action that can occur at that moment according to the tables, it must have been invoked, and must have the relevant guards controlling its execution set to true.

Repetition with a Boolean control mechanism.

In a similar manner to the modification made to selection we can place a guard to control repetition. Whilst the guard is true, repetition is permitted to occur. A syntax for the repetition could be:-

while (boolean expression) do element

where element is a subexpression of the path expression, and the boolean expression has the same properties as the guard of selections.

Example:

A writer may deposit k messages into a buffer. Each of these messages must be read by reader processes which accept only one message at a time. K may be of any size. The following type implements the synchronization:-

```
type distributor;
integer size;  size:=0;
. . .
path(deposit ; while size > 0 do remove)* end;
procedure deposit ( accepts integer value k,. . .):
     begin
     size:=k; (deposit messages);
     end;
procedure remove:
     begin
     size:=size-1; (removes message);
     end;
```

```
operations deposit, remove
endtype;
```

A process may execute deposit, passing the procedure a parameter k which is the number of messages it is depositing, Further deposits cannot be made until these messages are removed by remove and size has returned to zero, allowing the repetition to terminate.

Implementation of this repetition construction can be done using the same mechanism as that for selection. The element to be repeated is represented as having the boolean expression as a guard; the following sequence after the repetition has a guard which is the negation of the boolean expression.

Summary of modifications to selection and repetition constructions.

The modifications we have suggested are simple compared with the numerous possibilities that may be tried. With such constructions care must be taken to prevent deadlocks arising from erroneous values of the guards. The constructions provide a greater declarative power to the programmer using path expressions and provide a greater flexibility. The use of variables to control the flow of execution through a path expression allows much information to be recorded in the variables instead of incorporating it in some possibly large regular expression description of that information embedded in the path. Further research is required to establish the impact of such path notations on programming, and in particular whether such programs are readily comprehensible.

7.6 SUMMARY OF ALTERNATIVE PATH NOTATIONS.

The purpose of this chapter has been to illustrate that the path notation we have described is not unique and that there are many possible variations. Some of the variations are based upon adding additional constructions to the path notation described earlier. Amongst these, the parallel construction and the simultaneous execution have implementations when used with E-paths. Open paths demonstrate that there are other appropriate forms for expressing synchronization besides the regular expression based ones we have described in earlier chapters. Modifications were suggested to the selection and repetition construction and it was shown that path notations could be devised in which variables play an important role in the control of the synchronization of processes. This would allow greater freedom of expression than earlier path notations, but would involve greater complexity, additional error situations and a more complex implementation. In all the modifications and suggestions made in this chapter, the result has been to alter the declarative power of the individual path notations. Little attempt has been made to measure the scheduling or declarative power of these suggestions and this is a subject for future research. Finally, the Petri net model of a path expression is difficult to extend to other path notations which involve concepts of counting. In our view this difficulty is caused mainly by the lack of an abstraction mechanism for Petri nets which eliminates unnecessary detail.

CHAPTER 8

CONCLUSION.

Path notations are a new method of describing synchronization which provides a clear and structured approach to the description of shared data and the co-ordination of communication between concurrent processes. This method is flexible in its ability to express synchronization and may, for example, be used in a form equivalent in synchronizing capability to P and V operations (see Chapter 1).

The path expression describes synchronization between executions of procedures by processes. It is a statement of all permissible sequences of executions of the various procedures named within it. When combined with our type definition, path expressions provide a powerful tool with which to design shared data objects. The type contributes the protection necessary to avoid carefully designed synchronization schemes being upset by processes directly accessing the data. The type also collects together in one place all the implementation details of a shared object, and provides operations by which a user can manipulate the shared object. The path expression specifies the synchronization constraints needed to ensure the successful sharing of an object through the use of these operations. These constraints are expressed directly as permissible sequences of the operations and procedures which implement the type. This form of expression, in terms of whole operations or procedures, should be more readily comprehensible than prior methods for the specification of synchronization constraints. It removes the necessity of describing synchronization indirectly by embedding, within the text of the procedures, operations on a primitive object with known synchronization properties (for example, semaphores and P,V operations or conditions and signals and waits).

Just as there are many programming languages for describing processes, there are many possible path notations, and a choice of any one of them will be dictated by the use to which it is going to be put. A path notation based on regular expressions specifies the possible sequences of actions which may be allowed to occur. This form of expression has the advantage that, not only is it well known and theory exists about its use, but it has a structure which is easily learnt and used. Further, it is readily extendable by the addition of other synchronization constructions and by generalizations. The following results were obtained for path notations based upon regular expressions.

E-paths: A path notation which has a simple implementation is described by a set of production rules. The state machine corresponding to the regular expression generated from the productions is used to give a meaning to these expressions. It is shown that such state machines may be specified by an algorithm such that they are deterministic, strongly connected and minimal, and that they have no two transitions which share the same name. The restrictions imposed by the E-path notation on the set of regular expressions of which E-paths can be composed are sufficient to allow the synchronization for a procedure described in such an expression to be implemented by means of a P operation in the prologue and a V operation in the epilogue of that procedure. We have given a model for the interaction between E-paths and processes and shown that the implementation of E-paths is correct with respect to this model. In addition the model was used to help establish an equivalence relation between the scheduling power of a semaphore notation and E-paths. Finally, we gave a syntactic criterion which, for a class of paths and processes, was shown to be a necessary and sufficient condition to detect deadlock. This class is very restrictive, but does suggest that it might be possible to detect certain deadlocks using such syntactic criteria during the design and compilation of programs.

<u>R-paths</u>: R-paths are path expressions based upon an unrestricted regular expression specification of synchronization. We gave a model for the interaction between R-paths and processes, and R-paths were shown to have simulating nets which are safe and belong to the class of Simple Petri nets. This model was then used to introduce a new synchronization primitive in terms of which R-paths could be implemented simply. Another implementation was given in greater detail which describes a mechanism involving P, V operations, state tables and scheduling. It was shown that R-paths have greater scheduling and declarative power than E-paths.

<u>G-paths</u>: The G-path notation allows simultaneous description of the synchronization of an action by several path expressions, each constructed from a regular expression. This permitted

a synchronization specification to be factored into sequential components. Parallel execution by processes of different procedures named by this synchronization specification is possible if mutual exclusion is not implied by the synchronization constraints taken as a whole. G-path expressions are, however, more conceptually complex. For example, a G-path composed of R-paths may have dynamic properties. A procedure execution may be constrained by a particular synchronization specification of one of the constituent regular expressions of a G-path. It may also be controlled by one out of several alternative synchronization constraints in another regular expression of the G-path. A model for the interaction between G-paths and processes was given, and G-paths were shown to have simulating nets which are safe and belong to the class of General Petri nets. A syntactic criterion for detecting deadlock in a class of G-paths and processes was given, although again it is very restrictive. We gave an implementation for G-paths in terms of R-paths and it was thus shown that R-paths are equivalent to G-paths with respect to scheduling power. The second implementationwas given in more detail and uses incidence matrices of the abbreviated simulating net of the G-path, scheduling and P, V operations. Implementations of G-paths can be simplified if these G-paths consist only of constituent E-paths. We gave an implementation of GE-paths based upon the model of G-paths and processes using a synchronization primitive first described by Patil.

Path expressions may be based on more complex forms of expression than regular expressions:

Open path notation:

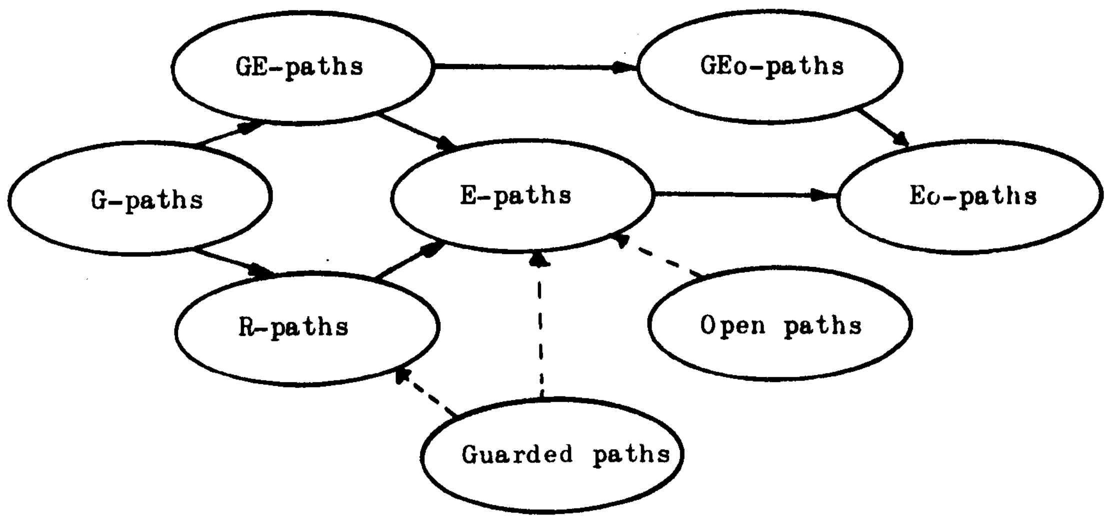
Open path expressions specify the *synchronization constraints* for a set of procedure executions, some of which may be concurrent. A set of processes may not only execute different procedures in an open path concurrently, but, unlike procedures synchronized by G-paths, they may execute the same procedure concurrently. The programming of synchronization with this mechanism is different in approach to that used for path notations based upon regular expressions. The occurrence of a procedure name in an open path does not necessarily impose any synchronization upon executions of that procedure by processes. Synchronization constraints are imposed upon the executions of procedures by embedding them within one of the synchronization constructions of the notation. Thus, the approach to specifying synchronization by means of open path expressions is based upon restricting the executions of procedures to an appropriate form. However, the approach to specifying synchronization by means of regular expression path notations is based upon describing the possible sequence of procedure executions which may be allowed to occur, and hence implicitly excluding those which are not required.

Repetition and Selection with a Boolean expression control mechanism:

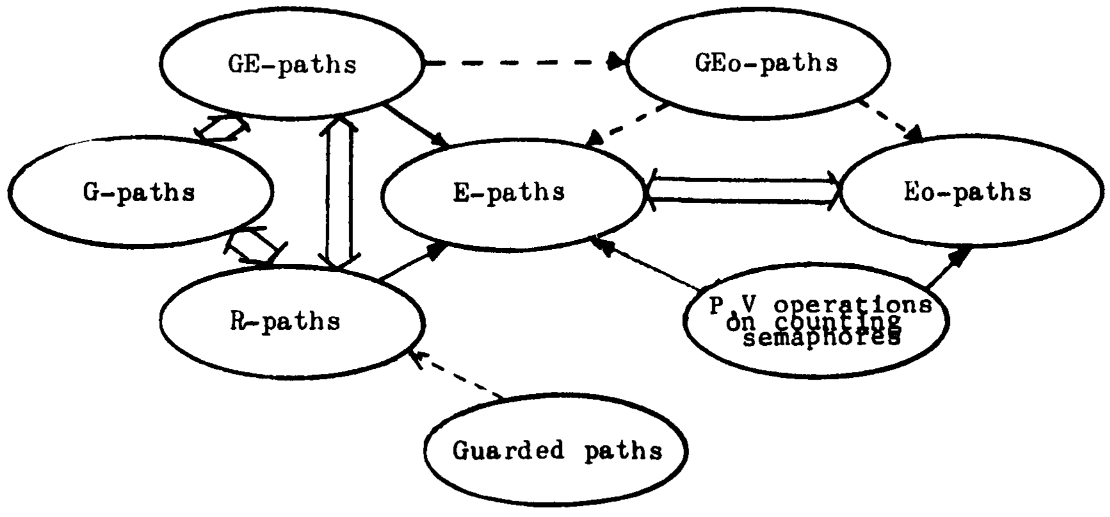
The guarded path notation is again very different from the previous notations. The synchronization specification depends upon the values of variables which may change dynamically with the execution of procedures named within the path expression. In previous notations we have attempted to isolate, as far as is possible, the synchronization specification from the specification of the processes. The use of these variables in the guarded paths implies a stronger dependency between the specification of the bodies of procedures and synchronization. The dependency is, however, introduced in a strictly controlled manner and may be of practical interest. Further research is required to investigate such notations. In particular, the ideas proposed above and the ideas proposed by Dijkstra would suggest that study of a unified approach to the specification of processes and synchronization (that is, the use of a similar notation for both) might yield significant results.

We have introduced two measurements of the power of synchronization of a path notation. The declarative power is a quantitative measure and the scheduling power is a qualitative measure of the synchronization which may be expressed by a path notation. It is to be expected that a program or specification of a system written using a path notation with a low measure of declarative power will make frequent use of procedures whose sole purpose is to allow further synchronization specifications to be imposed. Similarly, programs and specifications of systems using a path notation with a low measure of scheduling power may be expected to contain scheduling (that is, predicates on the timings of processes). Thus, the expected cost of using a path notation with poor declarative power is an increase in the number of procedures which need to be written, although this cost will be reduced if, for example, good macro facilities are available. The expected cost of using a path notation with poor scheduling power will be the number of scheduling algorithms which need to be written. However, this cost may be irrelevant in, for example, programs written to interface software with a machine and which will explicitly specify scheduling. In addition, these costs may be offset by the expense of implementing path notations which are more powerful. We can summarize the results of our analysis of path expressions with respect to power by the following diagram. Bi-directional arcs represent equivalences, and directed arcs represent a decrease in power following the direction of the arrow. Results which have been demonstrated in, or follow directly from, the thesis have been given solid arcs and dashed arcs represent conjectures which have not yet been established.

Declarative Power of synchronization.



Scheduling Power of synchronization.



Any scheme of co-operation and co-ordination amongst a set of concurrent processes can be used provided that these processes have the computational capability of a Turing Machine and there exists some simple mechanism which will arbitrate between these processes over the order in which they communicate with the mechanism. The design of these schemes is not easy and hence a variety of standard systems have arisen for specifying synchronization. Whether these systems have any practical value depends entirely upon how readily people may use and understand these systems. At present there is no reason to believe that P, V operations, monitors, or any other systems yet devised represent the best solution, and it would be presumptuous if we ceased research into other possible methods. I have shown that path expressions are an alternative method, in many ways more advantageous than preceding systems.

REFERENCES.

(Algol 68, 1968) Final Draft Report on the Algorithmic Language Algol 68.

(Bekkers, 1974) Y. Bekkers, A comparison of two high level synchronizing concepts. Report, Queen's University Belfast, Computer Science Department, 1974.

(Brinch Hansen, 1970) P. Brinch Hansen, Nucleus of a Multiprogramming System. CACM 13, 4, pp. 238-241, April 1970.

(Brinch Hansen, 1972) P. Brinch Hansen, Structured Multiprogramming. CACM 15, 7, pp. 574-578, July 1972.

(Brinch Hansen, 1974) P. Brinch Hansen, Concurrent Pascal; A Programming Language for Operating System Design. Information Science Technical Report No. 10, California Institute of Technology, April 1974.

(Campbell and Habermann, 1974) R.H. Campbell and A.N. Habermann, The Specification of Process Synchronization by Path Expressions. Lecture Notes in Computer Science (Editor G. Goos and J. Hartmanis), pp. 89-102, V16, Springer Verlag, 1974.

(Courtois, Heymans and Parnas, 1971) P.J. Courtois, F. Heymans, D.L. Parnas, Concurrent Control with "Readers" and "Writers". CACM 14, 10, pp. 667-668, October 1971.

(Dahl, 1970) O-J. Dahl, B. Myrhhaug and K. Nygaard, The Simula 67 Common Base Language. Norwegian Computing Center, 1970.

(Dijkstra, 1968a) E.W. Dijkstra, Co-operating Sequential Processes. In Programming Languages, F. Genuys (ed.), Academic Press, New York, pp. 174-186, 1968.

(Dijkstra, 1968b) E.W. Dijkstra, The "THE" Multiprogramming System. CACM 11, 5, pp. 341-348, May 1968.

(Dijkstra, 1973) E.W. Dijkstra, Hierarchical ordering of sequential processes. In Operating System Techniques, ed. C.A.R. Hoare and R.H. Perrott, Academic Press, 1973.

(Dijkstra, 1975) E.W. Dijkstra, Guarded Commands, non-determinacy and formal derivations of programs. CACM 18, 8, pp. 453-457, August 1975.

(Habermann, 1972) A.N. Habermann, Synchronization of Communicating Processes. CACM 15, 3, pp. 171-176, March 1972.

(Habermann, 1974) A.N. Habermann, On a solution and a generalization of the cigarette smoker's problem. Tech. Report, Carnegie-Mellon University, August 1974.

(Habermann, 1975) Path Expressions. Carnegie-Mellon Technical Report, 1975.

(Hack, 1972) M.H.T. Hack, Analysis of Production Schemata by Petri nets. Masters Thesis, MIT, February 1972.

(Harary, 1969) F. Harary, Graph Theory. Addison-Wesley, Reading, 1969.

(Harrison, 1965) Introduction to switching and automata theory, Mc Graw-Hill, New York, 1965.

(Hauck and Dent, 1968) E.A. Hauck and B.A. Dent, Burroughs' B6500/B7500 Stack Mechanism. AFIPS, SJCC, Vol.32, pp.245-251, 1968.

(Hoare, 1974) C.A.R. Hoare, Monitors: An operating system structuring concept. CACM <u>17</u>, 10, pp. 549-557, October 1974.

(Holt and Commoner, 1970) A.W. Holt and F. Commoner, Events and Conditions. Applied Data Research, New York, 1970.

(Hopcroft and Ullman, 1969) J.E. Hopcroft and J.D. Ullman, Formal Languages and their relation to Automata. Addison-Wesley, Reading, 1969.

(Lauer, 1972) H.C. Lauer, Correctness in Operating Systems. Ph.D. Thesis, Carnegie-Mellon, 1972.

(Lauer, 1974) P.E. Lauer, Petri nets with fewer tears . Unpublished manuscript, 1974.

(Lauer and Campbell, 1975) P.E. Lauer and R.H. Campbell, Formal Semantics of a Class of High-Level Primitives for Co-ordinating Concurrent Processes. Acta Informatica, 5, pp. 297-332, 1975.

(Lautenbach, 1973) K. Lautenbach, Exacte Bedingungen der Lebendigkeit für eine Klass von Petri-Netzen. Gesellschaft für Mathematik und Datenverarbeitung, Bonn, BMFT-GMD-82,1973.

(Lautenbach, 1974) K. Lautenbach, Dual Aspects of Process Co-ordination. Gesellschaft für Mathematik und Datenverarbeitung, Bonn, 1974.

(Lautenbach and Schmid, 1974) K. Lautenbach and H.A. Schmid, Use of Petri nets for Proving Correctness of Concurrent Process Systems. Proceedings of the IFIP Congress 74, North Holland Publishing Company, Amsterdam, pp. 187-191, 1974.

(Lipton, 1973) R.J. Lipton, On Synchronization Primitive Systems. Ph.D. Thesis, Carnegie-Mellon, June 1973.

(Myhill, 1957) J. Myhill, Finite Automata and the Representation of Events. WADC Tech. Rept., pp. 57-624, November 1957.

(Parnas, 1975) D.L. Parnas, On a solution to the Cigarette Smokers Problem (without conditional statements). CACM 18, 3, March 1975.

(Patil, 1971) S.S. Patil, Limitations and Capabilities of Dijkstra's Semaphore Primitives for Co-ordination amongst Processes. Project MAC, Computational Structures Group Memo 57, February 1971.

(Patil, 1975) S.S. Patil, An Asynchronous Logic Array. Project MAC Technical Memorandum 62, M.I.T., May 1975.

(Petri, 1962) Kommunikation mit Automaten. Shriften des IIM, Nr. 2, Bonn, 1962.

(Riddle, 1972) W. Riddle, The Modelling and Analysis of Supervisory Systems. Ph.D. Thesis, Stanford University, March 1972.

(Shaw, 1974) A. C. Shaw, The Logical Design of Operating Systems, Prentice-Hall 1974.

(Wegbreit, 1972) W. Wegbreit, B. Brosgol, G. Holloway, C. Prenner, and J. Spitzen, E.C.L. Programmers Manual. Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1972.

(Wirth, 1971) N. Wirth, The Programming Language PASCAL. Acta Informatica, 1, pp. 35-63, May 1971.