Implementation of Regular Path Expressions*

by
A. N. Habermann
Carnegie-Mellon University
Pittsburgh, PA 15213

Abstract

Path Expressions define sets of permissible operation sequences on typed objects. Path expressions specify process synchronization at a conceptual level instead of in terms of its implementation, which is the case if P,V operations or critical regions are hand-coded into the program text. A compiler takes care of translating path expressions into the necessary synchronization statements. This paper describes the compilation of regular path expressions which correspond to either deterministic or undeterministic finite state machines.

keywords and phrases: synchronization, process synchronization, synchronization primitives, programming constructs, parallel programming, concurrency, concurrent processes, abstract data types, compilers, monitors, finite state machines, finite automata, undeterministic finite automata

CR categories: 4.12, 4.13, 4.20, 4.22, 4.32, 4.35

---

*This research was sponsored by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 3597, and monitored by the Air Force Avionics Laboratory under Contract F33615-78-C-1151.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Table of Contents

1. Introduction

Path expressions were introduced in [1]. A path expression describes the possible operation sequences on a typed object. For instance,

path write; read **end**

specifies the permissible operation sequence on a communication buffer. This path allows an alternating sequence of *write* and *read* operations and disallows two or more successive *write* operations or two or more successive *read* operations. Compared to critical regions or P,V operations on semaphores, we observe that a path expression not only describes serial execution of its operands (usually referred to as *mutual exclusion*), but also specifies the *order* in which operations may be applied.

There are two other major differences between path expressions and critical regions or P, V operations on semaphores. First, a path expression is written separate from the program text. This has the advantage that the necessary synchronization is not buried in the programs, but is specified independent of the program text. Second, a path expression is a higher level concept in the sense that **while** and **for** statements are higher level concepts. Using P, V operations instead of **path** expressions is not unlike using **goto** instead of **while**. Indeed, a programming language can do without a looping statement if it has a **goto**, because loops can be constructed with conditional statements and **goto** statements! At this date is seems hardly necessary to repeat the arguments against this viewpoint.

Higher level constructs for looping are of course translated into "BRANCH" or "JUMP" instructions provided by the hardware. Path expressions must likewise be translated into lower level synchronization instructions. Fortunately, programmers are not concerned with these lower level concepts, nor with the translation, because this task is taken care of by a compiler. The subject discussed in this paper is the translation process of path expressions into lower level synchronization instructions.

Our discussion is restricted to regular path expressions. Such path expressions specify permissible execution histories, but do not allow parallel execution (as in the Readers, Writers problem [6]), nor do they allow arbitrary upperbounds on the number of executions of operations (no counters). After a short discussion of path operators and their representation, we consider where path expressions fit in programs for concurrent processes. Next we present an implementation which is closely related to P, V operations on semaphores. Then we discuss matters of reduction (which minimizes the number of states) and of nondeterministic path expressions. Finally, we show how a compiled path expression is used to enforce the specified execution histories.

2. Regular Path Expressions

Regular path expressions use three kinds of operators: ';' for sequencing, '+' for exclusive selection and '*' for indefinite repetition. The precedence of these operators is '*' (highest), ';', '+' (lowest). The precedence can be overridden by parentheses. For instance,

path a ; (b* + c) ; d* ; f **end**

consists of four *factors*: "a", "(b* + c)", "d*" and "f". The second factor consists of two *terms*: "b*" and "c". The third factor and the first term in the second factor are indefinite repetitions (indicated by the Kleene star). The delimiting pair of keywords "path" and "end" is equivalent to a Kleene star applied to the whole expression. Some execution sequences defined by this path expression are

a b b d d d f a b f ....
a b d f a c d d f ....

Observe that, following "a", either "b*" or "c" can be applied, but not both; "d" may be applied an arbitrary number of times between either "b*" and "f" or between "c" and "f" (including zero times). "b" may also be applied zero or more times.

Following the convention of common algebraic notation, we often omit the ';'. The given example then reads like this

path a (b* + c) d* f **end**

A regular expression can be represented by a finite state machine [6]. A FA which reflects the permissible sequences defined by the path expression above is

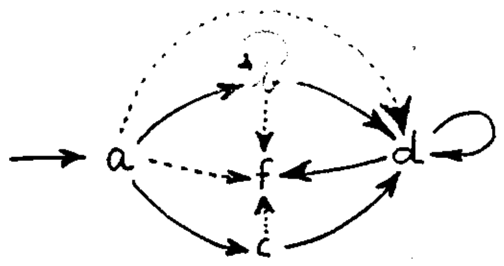

Figure 1. Example of a FA representation.

A state represents a path element and its position in that path. The sequencing is expressed by the arcs. There is no basic difference between solid and dotted arcs, both represent sequencing information. The dotted arcs in the picture reflect the possibility that "b*" and "d*" are executed zero times. They were put in by applying the construction rule that a state, followed by a starred element, must point to all successors of that element. An application of "b" or "c" may directly be followed by an application of "f", even "a" may be directly followed by "f".

3. Path Expression Declarations

Path expressions are based on the idea that the need for serializing applications of operations on a shared object is a property of that *object itself* instead of a property of the *users* of this shared object. It is therefore natural to find a path expression in the *definition* of a data object and *not* in the programs of the users of that object. Taking the current common view on data abstraction (which originated in Simula67 [2] and has been refined in languages such as CLU or Alphard [5, 8]), the natural place for writing a path expression is in the data section of an abstract data type definition. An abstract data type definition has the general format

```
type name(params) : t =
     data section
     initialization section
     data operation section
end type
```

Including it in the data section of a type t, implies that we consider a path expression to be part of the data record representing objects of type t. For example,

```
type segment (N : int) : s =
     array 1 .. N of word ;
     path fetch ; (read + write)* ; save end
     init ....
     let A : array of word in
     s.fetch = "make segment s accessible"
     s.read(A) = "copy words from s into A"
     s.write(A) = "copy words from A into s"
     s.save = "make segment s inaccessible"
end type
```

The definition of a path expression is uniform for all data objects declared of a type t. However, each object of type t has its own instantiation of that path. This instantiation is created when the object is declared. A path expression controls the sequence of operations on *individual* objects of type t and not those of the *total collection* of objects of type t. One segment x may be in a state in which it can be fetched, but not written or read, while some other segment y can be written, read or saved. Operations on different objects of type t may overlap in time; mutual exclusion is enforced per individual object by its private instantiation of the path expression.

4. The Implementation

There is a strong similarity between the path expression implementation and P,V operations on semaphores, but there are also some significant differences. The record field which represents a path instantiation is called a *path variable* ("pvar" for short). It corresponds to a semaphore and has three sub-fields, a *mode* sub-field, a *state* sub-field and a *waitinglist* sub-field. The mode field has the value "busy" or "free", depending on whether or not a path element is being executed. The interpretation of the state field depends on the value of the mode field. If *mode* is "busy", *state* reveals the executing operation and its position in the path expression. This corresponds to a particular state in the FA representation. If *mode* is "free", the value of *state* represents the set of permissible successor states. This corresponds to the set of arcs in the FA leaving the most recent state. These arcs determine the set of operations currently eligible for execution (the ones that may "fire").

Example.

path a b (a + b*) c **end**

If the first occurrence of "b" is the most recently executed operation and *mode* is "free", then *state* = {second a, second b, c}. This is the set of operations currently permitted to fire. If "b" is executed again, *mode* will be "busy" and *state* = second b.

The state field corresponds to the value field of a semaphore and the waitinglist field corresponds to a semaphore waitinglist pointer. The latter points to the list of processes waiting for *state* to change to a value which permits execution of the attempted operation. Note that each object of type t has its private path variable and, consequently, its own waitinglist (see Figure 2). (These waitinglists are usually implemented by a unique link field in each process control block.)

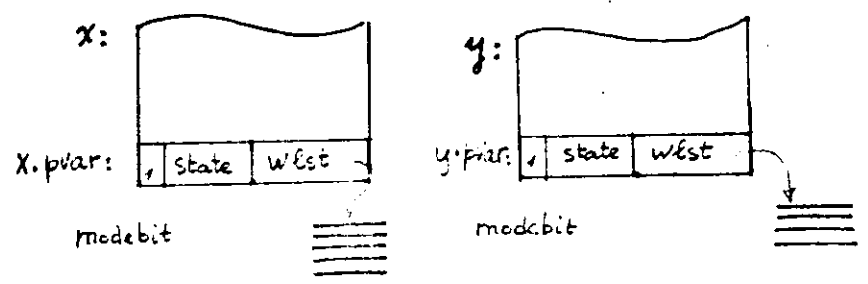

Figure 2. Implementation of pvars.

Let some type definition t contain a path expression which has f, g, p and q as its operands. The programs for f, g, p and q are defined in the operation section of t. In order to get the desired effect of the path expression, the compiler processing the type definition places a synchronization statement at the beginning of the body of each operation and one at the end. These synchronization statements are called "pro" (for *prologue*) and "epi" (for *epilogue*). Each operation body is embedded in a *pro*, *epi* pair. For our example, we write

```
F := x.pro(fval) ; "body of f" ; x.epi
G := x.pro(gval) ; "body of g" ; x.epi
P := x.pro(pval) ; "body of p" ; x.epi
Q := x.pro(qval) ; "body of q" ; x.epi
```

where x is a formal representing the object to which f, g, p or q is applied.

The argument in *pro*-calls is generated by the compiler and is operation specific. It reflects the set of states in which an operation is permitted to fire. (Further details are discussed in Section 7.)

Let us see what happens in a running system. Execution of a *pro*-call is very similar to executing a P operation. However, instead of testing for a positive semaphore value, *pro* compares its argument with the current value of x.pvar.state, where x is the object to which the operation is applied. If x.mode is "free" and x.pvar.state permits the attempted operation, *pro* sets x.mode to "busy", x.pvar.state to the selected state and terminates. The attempted operation is then applied to object x. If x.mode is "busy" or x.pvar.state does not permit the attempted operation, the running process is put on the waitinglist which is accessible through x.pvar.wlst.

The *epi*-call at the end of an operation changes x.pvar.state to the set of successors that are now permitted to fire. If some process is waiting for this state, *epi* reactivates one of these processes (which can now complete its *pro*-call); otherwise, *epi* sets x.mode to "free". Schematic programs for *pro* and *epi* are given in Section 7.

A major difference between pvars and semaphores is in their use. Semaphores are incremented or decremented, whereas pvars are assigned to. The latter is the crucial mechanism that allows us to discriminate between operations. A pvar value allows certain operations to fire while others will be delayed. A semaphore is not able to discriminate in this way. When a semaphore value is positive, any operation can pass a P operation applied to that semaphore.

Assigning to semaphores is very dangerous, because programmers write semaphore operations directly in their programs. This is not so for pvars. The *pro* and *epi* calls are not written by a programmer, nor does the program compute the argument passed in a *pro* call. Instead, the compiler generates these calls and derives the argument values from the given path expression. Assigning to pvars in *pro* and *epi* is harmless, because this feature is not available to programmers.

Management of pvar waiting lists differs from that of semaphore waiting lists. The information in the list is different and processes are selected differently. Conceptually, a pvar waitinglist entry consists of a process identification and the argument value of the incomplete *pro*-call. (A semaphore waitinglist entry consists of nothing more than a process identification.) When pvar.state is changed by an execution of *epi*, the waitinglist is searched for a process waiting for this new state value. It is possible that none of the waiting processes is waiting for this particular value of pvar.state. Thus, it may happen that *epi* reactivates none of the waiting processes and sets pvar.mode to "free" while the waitinglist is not empty! Such a situation is unthinkable for semaphores. Each process on a semaphore waitinglist is eligible to continue. This we don't want for pvars! It is not impossible that certain operations are still not allowed to proceed although the state has changed. It may be necessary that other operations execute before the state changes to the required value.

5. Reduction

For a realistic implementation, the amount of information which must be remembered for a path expression should be as little as possible. This is achieved by reducing a given path expression to an equivalent one which has fewer operands. An example of two equivalent paths is

path a(p + q) + b(q + r) + c(p + r) + (a + b)r + (b + c)p + (a + c)q **end**
path (a + b + c)(p + q + r) **end**

A simple reduction scheme is based on searching for common subexpressions and elimination of superfluous terms. This scheme makes use of the fact that

$$ab + ac \rightarrow a(b + c)$$
$$ac + bc \rightarrow (a + b)c$$
$$a + b \rightarrow b + a$$
$$a + a \rightarrow a$$

This scheme was used to reduce the longer path expression into the shorter one of the example above.

If this algorithm is able to reduce a given path expression into one which has fewer than N operands (say N = 32), it is probably unnecessary to put more effort into further reduction.

Further reduction is certainly possible, because reduction of finite state machines is a solved problem. There is an algorithm which reduces any given FA to an equivalent one which has a minimum number of states [4]. Applied to path expressions, the algorithm works as follows.

1. Draw the FA representation of the given path expression.
2. Determine for each state the set of successor states.
3. Assign to each state a "class" attribute which is the set of successor names.
4. Put all states with the same attribute in a class and give that class a name.

```
loop
   with each state do
      5.  new attribute(state) := {class names of successors}
   end with

      6.  if some states in one class have different new attributes
        6.1  set attribute(state) to new attribute(state)
        6.2  put states with same attribute in one class
        6.3  give these new classes unique names
        6.4  repeat loop
      end if
end loop
```

7. Merge all states that have the same class attribute and the same name into a single state. (This is accomplished by redrawing all their incoming and outgoing arcs to and from that single state.)

It is not difficult to show that this algorithm terminates. We see that step 6 repeats the loop only if it increased the number of classes. The upperbound of this number is the number of states. The proof that this algorithm produces the minimum state FA is given in [4, page 29]. We apply the algorithm to path expression

path a(b + b(ab)*) **end**

to show that it can do more than common subexpression elimination. The initial FA for this expression is

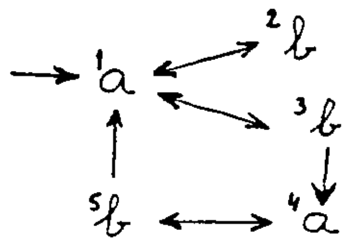

Figure 3. Example of a FA with redundant states.

Application of the reduction algorithm results in

state	**name**	**succ**	**{name(succ)}**	**class$_0$**
1	a	2 3	{b}	B
2	b	1	{a}	A
3	b	1 4	{a}	A
4	a	5	{b}	B
5	b	1 4	{a}	A

The initial classification is

A = {2, 3, 5} , B = {1, 4}

When the loop is executed for the first time, we find

{classnames(succ(2))} = {classnames(1)} = {B}
{classnames(succ(3))} = {classnames(1, 4)} = {B}
{classnames(succ(5))} = {classnames(1, 4)} = {B}

and

{classnames(succ(1))} = {classnames(2, 3)} = {A}
{classnames(succ(4))} = {classnames(5)} = {A}

The loop is not repeated, because all states in both classes map into the same class name. Thus, step 7 merges states 1 and 4 into one state and also states 2, 3 and 5. The result is a reduced FA with only two states! The path expression corresponding to the reduced FA is

path ab **end**

The common subexpression algorithm is not able to achieve such a drastic simplification. Practice has shown that path expressions are usually not very complicated. This and the fact that the reduction algorithm is applied at compile time (rather than at execution time) make that the $O(n^2)$ complexity of the algorithm causes no serious problems.

6. Nondeterministic Paths

A path expression is *nondeterministic* if two or more successor states in its FA representation have the same name. An example of an nondeterministic path expression is

path a $(fg)^*$ b^* $(fg)^*$ c **end**

Its FA representation is

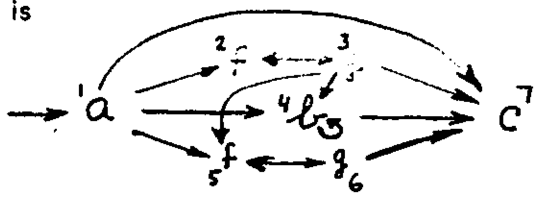

Figure 4. Example of a nondeterministic FA

Nondeterminism is caused by the successor states "f*" of states "a" and "g". It is well known that a nondeterministic FA can be transformed into an equivalent deterministic FA [4, pp 31-33]. A nondeterministic path expression is transformed into an equivalent deterministic path expression by the transformation algorithm

```
1. Draw the nondeterministic FA representing a given path.

2. Make a state table, listing (number, name, successors, new successors).
      All new successor sets are initially empty.

3. Initialize collection C of synonymsets to empty, where a
   synonymset contains all states in a successor set with a common name.

with each state s do
  4. initialize new successor set to successor set.

  with each synonymset y that has more than one element do
    5. if y ∈ C, lookup number(y)
       otherwise
            5.1 add new state to table.
            5.2 let number(y) be number(new state).
            5.3 set name(new state) to common name of states in y.
            5.4 set succ(new state) to UNION(succ(states of y)).
            5.5 add y to C.
       end if

    6. remove y from new successor set.
    7. add number(y) to new successor set.
  end with
end with

8. remove unreachable states.
```

The with statements terminate, because an upperbound for the number of synonymsets is the number of all possible subsets of states initially entered into the table in step 2. Step 8, "remove all unreachable states", needs some explanation. We add a unique initial state to the given path expression before we construct the table. Adding this initial state transforms

path ... **end** into **path** u; (...) **end**

The table is constructed for this modified path with state(u) = 0. In order to preserve the sequencing of the given path expression, all occurrences of state 0 in a successor set are replaced by the successor set of state 0. The set of all reachable states is defined as the closure under *successor* of successor(0). That is, we start with the successor set of state 0. Then we keep including all successors of the states that we collected until no more new states can be added. All states not in this collection are unreachable.

Applied to the example above, the table for the modif.ad path is

state	name	{succ}	{new succ}	C
0	u	1	1	
1	a	2 4 5 7	4 7 8	
2	f	3	3	unreachable
3	g	2 4 5 7	4 7 8	unreachable
4	b	5 7	5 7	
5	f	6	6	
6	g	5 7	5 7	
7	c	$0 \rightarrow 1$	1	
8	f	3 6	9	{2 5}
9	g	2 4 5 7	4 7 8	{3 6}

State 8 is added when synonymset y = {2 5} is found in succ(1) and state 9 when synonymset y = {3 6} is found in succ(8).

States 2 and 3 are unreachable when we switch to the new successor states, because closure(new state 0) = {1 4 7 8 5 6 9}. The deterministic path expression equivalent to the given one is

path a $(fg)^*$ (c + bb^* $(fg)^*$ c) **end**

7. Path Expression Dynamics

We have seen how path expressions are simplified, how paths are instantiated as part of the data representation of typed objects and how data type operations are embedded in a *pro,epi* pair. In this section we discuss how the implementation assures that only those histories which are specified by a path expression can be applied.

The permissible operation sequences specified by a path is *type specific* information. It is uniform and fixed for all objects of a type. Therefore, there is no need to store this information with every object. The semantics of a particular path expression is *own* data of a type and is shared by all objects of that type in the same way that code of data operations is shared.

The sequencing information specified by a path expression p is stored in two sets of vectors, *successor* vectors (one for each state) and *permission* vectors (one for each operation mentioned in p). A successor vector succ(s) contains all successor states of state s. Successor vectors are precisely the ones generated in the reduction algorithm and the deterministic transformation algorithm. A permission vector perm(f) contains all states in which operation f is permitted to fire. The permission vectors are easily derived from the tables used in the transformation algorithm or the reduction algorithm.

Example. **path** write (write (write read)* read)* read)* **end**

state	**{succ}**	**state**	**{succ}**	**oper**	**perm**
1	{2 6}	4	{3 5}		
2	{3 5}	5	{2 6}	write	{1 2 3}
3	{4}	6	{1}	read	{4 5 6}

We discussed in Section 4 that the body of an operation f mentioned in a path is embedded in a *pro, epi* pair. The argument of the *pro* call preceding the body of f is the permission vector perm(f). This argument is used in *pro* to determine whether or not operation f is allowed to fire in the current state. Permission is granted if and only if *mode* = "free" and one of the permissible states indicated in pvar.state matches one of the states in perm(f). The latter test is performed in *pro* by an AND operation applied to pvar.state and argument perm(f).

A schematic program for *pro* executed by some process P is

```
x.pro(u : permission vector) =
    with x.pvar do
        if mode = "busy" or AND(state,u) = empty,
            put (P, u) on wlst; halt P
        end if
        mode := "busy"; state := AND(state, u)
    end with
```

pro sets pvar.state to the matching state. This state value is used by *epi* to set state to succ(state) at the end of the operation. A schematic program for *epi* is

```
x.epi =
    with x.pvar do
        state := succ(state);
        if (∃P) AND(state, u_P) ≠ empty,
            remove(P from wlst); reactivate(P)
        else mode := "free"
        end if
    end with
```

Remarks

- *pro* and *epi* are both implemented as indivisible operations in order to avoid race conditions when several processes operate on a common object [3].
- The name "AND" is used on purpose to suggest an efficient implementation in terms of bit vectors. If the number of states does not exceed the wordlength of the given hardware, the vectors can be implemented as single words. The "AND" operation applied to machine words is very efficient.

Let us see how pvar.state changes for a three-slot communication buffer described earlier in this section. We find perm(write) = {1 2 3} and perm(read) = {4 5 6}.

state(free)	**operation**	**pro : AND**	**epi : succ**
{1}	write	{1}	{2 6}
{2 6}	write	{2}	{3 5}
{3 5}	write	{3}	{4}
{4}	write	no match	write cannot go
{4}	read	{4}	{3 5}

The last *epi* will reactivate the unsuccessful fourth write operation, because perm(write) matches the last successor state.

A slightly different implementation of *epi* makes it possible to handle nondeterministic paths without first transforming them to deterministic paths. The difference between the two kinds shows up in *pro* when the current state is set. In case of a nondeterministic path, the result of AND(state, permission vector) may be a set of states instead of a single state. If this happens, *epi* must set state to the union of successors of all these states, because the path may be continued along an arc of anyone of the current states. This is achieved by replacing the first statement in *epi*

"state := succ(state)" by "state := OR(succ(all current states))"

This point is elucidated by tracing the history "a f g c" for the nondeterministic path expression of Section 6.

path a (fg)* b* (fg)* c **end**

for which

perm(a) = {1}, perm(b) = {4}, perm(c) = {7}
perm(f) = {2 5}, perm(g) = {3 6}

states	**operation**	**pro: AND**	**epi: OR**
{1}	a	{1}	{2 4 5 7}
{2 4 5 7}	f	{2 5}	{3 6}
{3 6}	g	{3 6}	{2 4 5 7}
{2 4 5 7}	c	{7}	{1}

Leaving a path expression in its nondeterministic form is preferable if the number of states increases beyond the machine's wordlength by transforming it to a deterministic path. Successor vectors and permission vectors are not representable by single machine words if the number of states exceeds the machine's word length. An argument against leaving a path in its nondeterministic form is that the OR operation in *epi* is executed at runtime, whereas the transformation algorithm is executed at compile time. Thus, runtime efficiency is enhanced by using the deterministic version.

8. Conclusion

Regular Path Expressions can be compiled into simple synchronization statements operating on the pvar of a shared object. The synchronization statements are very similar to P, V operations and pvars correspond to semaphores. Some differences are caused by the fact that path expressions specify the order in which operations can be applied. It is therefore normally the case that at a given time not every path operand can be applied. The specified execution histories are enforced by the implementation.

A path expression can be transformed into a deterministic path which has a minimum number of states. A by-product of applying the reduction algorithm is the construction of all successor vectors and permission vectors. This information is part of the type definition and shared by all objects of a type. It is used by the synchronization statements *pro* and *epi*.

The implementation handles nondeterministic path expressions as well as deterministic ones. At ambiguous points, a pvar reflects a set of states instead of a single one. There is a little more overhead in dealing with nondeterminism, because the set of successor states is then constructed by taking the OR of several successor vectors.

References.

1. Campbell, R. H. and A. N. Habermann,
"The Specification of Process Synchronization by Path Expressions,"
Lecture Notes in Computer Science Vol. 16 (eds. G. Goos and J. Hartmanis),
Springer-Verlag, 1974, 89-102.

2. Dahl, O. J., E. W. Dijkstra, C.A.R. Hoare, Structured Programming,
Academic Press, London and New York, 1972.

3. Habermann, A. N., Introduction to Operating System Design
Science Research Associates, Palo Alto, Calif (March 1976)

4. Hopcroft, John E. and Jeffrey D. Ullman, Formal Languages and
Their Relation to Automata, Addison-Wesley, 1969.

5. Liskov, Barbara, Alan Synder, Russell Atkinson and Craig Schaffert,
"Abstraction Mechnaisms in CLU," Proceedings of an ACM Conference
on Language Design for Reliable Software (ed. David B. Wortman),
March 1977, 166-178.

6. Minsky, Marvin, Computation: Finite and Infinite Machines,
Prentice-Hall, Inc., Englewood Cliffs, NJ, 1967.

7. Courtois, P. J., R. Heymans and D. L. Parnas,
"Concurrent Control with 'Readers' and 'Writers'"
Comm. ACM, 14, 10 (Oct 71)

8. Shaw, Mary, W. A. Wulf and Ralph L. London,
"Abstraction and Verification in ALPHARD: Defining and Specifying Iteration,"
Proceedings of an ACM Conference on Language Design for Reliable Software
(ed. David B. Wortman), March 1977, 153-165.