NOTICE WARNING CONCERNING COPYRIGHT RESTRICTIONS:
The copyright law of the United States (title 17, U.S. Code) governs the making of photocopies or other reproductions of copyrighted material. Any copying of this document without permission of its author may be prohibited by law.

PATH EXPRESSIONS

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

June 1975

Abstract. Traditionally, synchronization of concurrent processes is coded in line by operations on semaphores or similar objects. Path expressions move the responsibility of implementing such restrictions from the programmer to a compiler. The programmer specifies as part of a type definition which execution sequences are permitted. The advantage of using path expressions instead of P, V operations on semaphores (or similar operations) is comparable to the advantage of using **for**- and **while**-statements instead of JUMP or BRANCH instructions. In this paper the rules for writing a path expression are described, parsing and implementation are discussed and the use of path expressions is shown by a number of examples.

This work was supported in part by the Defense Advanced Research Projects Agency under contract F44620-73-C-0074 monitored by the Air Force Office of Scientific Research, and in part by the National Science Foundation under grant DRC74-24573.

1. INTRODUCTION

The concept of subroutine was invented to save the programmer the unrewarding task of rewriting the same lines of code several times. Presently, the significance of procedures or functions goes far beyond the original subroutine idea. The procedure declaration is an important program design tool. First, it allows the programmer to split the programming task into several parts, where each part is significantly smaller than the total program. Secondly, the procedure concept provides an important abstraction tool. In a well-designed program, the **implementation** of a procedure is irrelevant to the program environment in which the procedure is called. All that matters at the call site is the **functional specification** of the procedure, i.e. the parameters it expects and its effect on the calling environment.

In large programs (such as a compiler or an operating system) the procedure concept is useful but not sufficient to make the programming task simple enough. Here the number of procedures rises to such a height that it becomes necessary to partition the set of all procedures into meaningful subsets. The promising concept for achieving such a meaningful partitioning is that of a "type definition" (or "class" in SIMULA 67). A type definition describes the internal structure of a set of data objects and all the procedures which define operations on these objects. E.g., applied to compiler design, one finds type definitions for objects such as a hash table, a symbol table, a lexeme, a syntax stack, etc. More detailed examples follow in subsequent sections.

An operating system makes it possible that user programs share resources and run in parallel. However, it is a well-known fact that user programs cannot have unrestricted access to shared objects [1]. In many cases only one operation on a

shared object may be executed at a time, though the order is immaterial. In other cases operations must be executed in a given order (e.g. placing the first message in a queue must precede taking a message out of the queue).

Until now, concurrency restrictions have been coded in line by inserting critical regions and wait/signal operations in the programs [2]. There has been an extensive discussion about a variety of synchronization primitives. An analysis of their relative power is found in [3]. Path expressions do not introduce yet another synchronization primitive. A path expression relates to such primitives as a **for**- or **while**-statement of an ALGOL-like language relates to a JUMP or BRANCH instruction in an Assembly language. A programmer specifies control by a while-statement; the statement is implemented by test and branch instructions. Likewise, a programmer specifies restrictions on the execution of operations on shared objects; the specified restrictions are translated by a compiler into instructions which use synchronization primitives. The purpose of writing path expressions is to bring the design of concurrency restrictions to a higher level in the same sense as eluded to by the phrase "higher level programming language". Programmers have learned by experience how important this is.

Concurrency restrictions apply to operations which access a shared object. Since a shared object is completely described by a type definition, a path expression is placed in a type definition as part of the internal structure description (examples follow in subsequent sections). A path expression describes the allowable sequences of executing operations on a shared object. In the common simple case, a path expression is a regular expression from which all possible execution sequences can be derived.

1. INTRODUCTION

Subsequent sections deal with simple path expressions, with conditional elements and concatenation of path expressions, and with parsing and implementation of path expressions. The use of path expressions is demonstrated by several examples, some of which have been borrowed from the paper on Monitors [4].

2. ON WRITING PATH EXPRESSIONS

2.1 A path expression is delimited by the keywords **path** and **end**. Its operands are function names. The operators are (in precedence order) *, ;, +. The precedence is overruled by parentheses ( ). The operator ; is the sequencing operator. The sequencing operator can be omitted (analogous to the multiplication operator in arithmetic expressions). E.g.,

path a ; b ; c **end** or **path** a b c **end**

means that the only permissible execution sequence is : a b c a b c a b c a b c a b c a b c ----. An operand to which the sequencing operator applies is called a "factor".

The keywords **path, end** represent an implicit Kleene star, i.e. once the end of a path is reached, the path can be entered again at the beginning. The operator * also represents the Kleene star. It is used as a unary postfix operator indicating that the operand it modifies can be executed zero or more times before going on to the next. E.g.,

path p ; (q ; r)*; s **end** or **path** p (q r)* s **end**

means that an arbitrary number of sequences q; r (including none) can be executed in between an execution of p and the subsequent execution of s.

2. ON WRITING PATH EXPRESSIONS

The operator + represents exclusive selection. E.g.,

path f ; (g + h) ; k **end**

means that either a g or an h (but not both or none) must be executed between an execution of f and the subsequent execution of k. An operand to which the operator + applies is called a "term".

The operator ; is distributive with respect to the operator +. E.g.,

path f (g + h) k **end** = **path** (fg + fh) k **end** = **path** fgk + fhk **end**,

because in all cases an execution of k is separated from the preceding execution of f by an execution of either g or h.

The operator * is not distributive with respect to either + or ;. E.g.,

path p ; (q ; r)* ; s **end** ≠ **path** p ; (q* ; r*) ; s **end**

because in the latter all q's between a p and an s precede all r's between these two. Also,

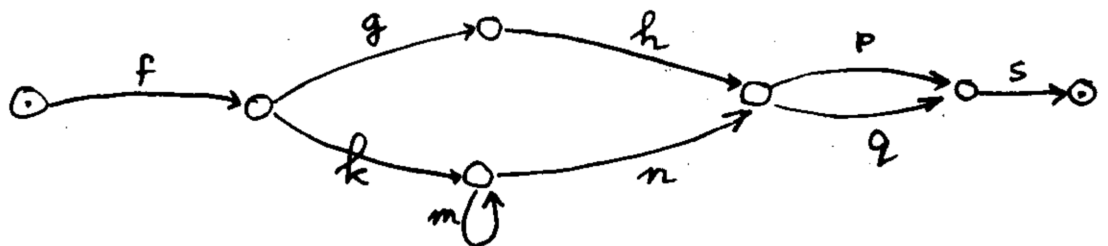
path f (g + h)* k **end** ≠ **path** f (g* + h*) k **end**

because the first path allows an arbitrary mixture of g's and h's between every pair (f,k), whereas the second path allows either all g's or all h's (but no mixture) between an execution of f and a subsequent execution of k.

2.2. A path expression can easily be translated into a graph model representing the finite state machine defined by the regular expression. The arcs in the graph represent the functions, the nodes represent the initial state, the final state and the sequential states corresponding to the semicolons in the path expression. E.g.,

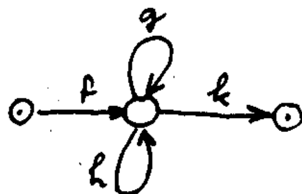
path f (gh + km*n)(p + q) s **end**

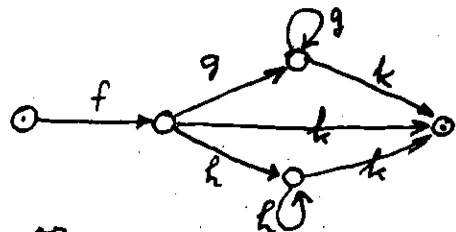
is represented by the graph



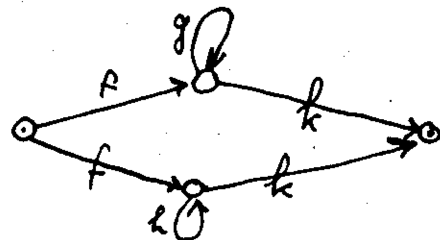
(The final state is identical to the inital state.)

The difference between **path** f (g + h)* k **end** and **path** f (g* + h*) k **end** is shown in the graphs below.





or



The first of these paths is called a "simple path". The second is not a simple path. In terms of the graphs, a simple path has a graph in which no two arcs carry the same name. In terms of finite state machines, an operand of a simple path has a unique starting state and a unique result state. The result state is in general a function of the current state and the executed function. However, the result state in a simple path expression is a function of the executed operation, but not of the current state. It was shown in [5] that simple path expressions can be implemented by P,V operations on Boolean semaphores.

2.3. In many cases in which synchronization is necessary, simple paths are adequate. E.g., a set of critical regions {a,b,c,d} is programmed by

```
path a + b + c + d end
```

because this path specifies that each time exactly one of the four functions can execute.

If the execution of a function named in a path expression is attempted and the current state of the path expression does not allow its immediate execution, the program attempting the execution is suspended. When the state of the path expression changes and some programs are waiting for a function which can be executed in that state, the longest waiting program will be reactivated and will be enabled to execute the requested function. In other words, the programs are scheduled per state in first-come, first-serve order (FCFS). In Section 3 we will see how the order can be specified, up to a limited extent, by the programmer.

Example 1. A communciation between two processes is initiated by declaring a buffer which can hold a message whose interpretation is known to both processes. Assuming the existence of the **type** message, the buffer objects are defined by

```
type oneslotbuffer =
        var mes = message
        path deposit; remove end
let b = oneslotbuffer, m = ref message in
   op b.deposit(m) = mes ← m
   op b.remove(m) = m ← mes
end
```

The type definition consists of two parts. The first part is a record describing the internal structure of the objects of that type. The second part describes the operations which can be performed on these objects. The **let**-clause specifies the parameters used in the operations. The prefix parameter of an operation is of the same type as the type in which the operation is declared (SIMULA 67 laudatur). Unprefixed fieldnames, such as "mes", in the body of an operation relate to the given prefix parameter.

The internal structure defined in a type definition can be accessed in the programs of the operations defined in that type. Outside the type definition, the operations can be applied, but the internal structure is not accessible.

The path expression is part of the internal structure, i.e., every object declared of type oneslotbuffer has its own path. A path is not defined for the collection of objects of that type; on the contrary, a new instance of the path is created every time a new object of that type is declared.

The path expression specifies that every deposit must be followed by a remove action and every remove by a deposit. If a second remove is attempted, it will automatically be delayed until another deposit has taken place. A second attempt to deposit is likewise delayed until the first message has been removed.

Example 2. Some cases which an operating system must handle call for a scheduling discipline different from the straightforward FCFS discipline postulated for path expressions. An example is given in [4] suggested by A. Ballard and J. J. Horning.

An alarm clock service must be designed which enables a calling program to delay itself for a given number of time units, or "ticks". A program sets the alarm clock by calling wakeme(n = integer). The programs must be awakened by smallest wakeup time first and not in FCFS order. Time is measured by a hardware clock which activates the alarm clock procedure "tick" every time unit.

The alarm clock feature is provided by a definition of "wakeuptime" and a definition of "alarmclock".

```
type wakeuptime =
   var wt = integer (∞)          comment  wt is initialized with the value ∞
   path set; pass; wakeup end
   let u = wakeuptime, n = integer in
      op u.set(n) = wt ← n
      op u.pass = wt ← 0
      op u.wakeup = wt ← ∞
      op u.val = result integer; return wt
end
```

A type definition describes an object and its operations, but it does not declare any objects of its type. If we wish to combine a type definition with the declaration of one object of that type, we use the keyword **decl** instead of **type**.

The alarm clock maintains a list of wakeuptimes. A list w is declared by

```
var w = list <n> of <t>
```

where n is the number of initial elements and t the type of the list elements. The current element of a list w is represented by .w. Relevant list operations are

advance w — current is set to the next element or to nil if it is moved past the end of the list

reset w — set current back to the first element

new .w — a new list element is created and inserted preceding the current. The current is set to the new element

free .w — the current element is deleted and current is advanced

Assuming that only one alarm clock is needed, the declaration of alarm clock is

```
decl alarmclock =
   var wlist = list 1 of wakeuptime
comment the list is initialized with a permanent last element with value ∞
   var now = integer(0), first = integer(∞)
   path setalarm + tick end
let n = integer in
   proc setalarm(n) = result ref wakeuptime
   begin cons t = n + now
      reset wlist; while  .wlist.wt < t do advance wlist od
comment termination is guaranteed by the fact that last element.wt = ∞
      if first > t then first ← t fi; new.wlist; .wlist.set(t)
      return ref .wlist
   end
comment a proc is not available outside a type definition
```

```
op wakeme(n) =
begin var x = setalarm(n); x.wakeup end
op tick =
begin now ← now + 1; reset wlist
   while .wlist.wt ≤ now do .wlist.pass; free .wlist od
end
comment tick is activated at regular intervals by the hardware clock
```

A program calling wakeme adds a new element to the list of wakeuptimes which is inserted such that the list is sorted by ascending wakeup times at all times. The program then applies wakeup to this element. The path expression in **type** wakeuptime ensures that the wakeup operation is not scheduled until operation pass has been applied to this element. The latter operation is performed by tick, but not until "now" overtakes the stated wakeuptime.

This solution seems more complicated than the Monitor solution given in [4]. This is primarily due to explanation of the list operation. However, the given solution deserves this title more than the Monitor solution. The latter has the drawback that the program whose wakeuptime is the first is awakened every clock tick! (Imagine the poor guy who wants to get up early in the morning at 5:30 a.m. and turns in early at 9:00 P.M. He is awakened after every time unit and he must inspect his watch every time to see if it is time to get up.) A fair comparison cannot be made unless an accurate Monitor solution is presented.

Example 3. The delay between two data transfers from (or to) a disk with a moving head is proportional to the distance the head must travel. Therefore, the most efficient schedule for processing disk requests is not FCFS, but "nearest track first",

i.e., if a transfer involving track t is completed, the disk scheduler should pick as next request the one asking for a track nearest to t. However, this scheduling discipline has the drawback of a potential starvation. It may happen that the scheduler picks requests for tracks at one end of the disk all the time, neglecting requests for the other end of the disk. This problem is solved by an "elevator schedule". The scheduler will pick the nearest track, but it will move in one direction, either up or down like an elevator, until there are no more requests for tracks in that direction.

A program activates the disk by placing a command in its command buffer. We assume the existence of **type** command, describing the internal structure of a disk command. A disk device is represented by the definition

```
type DISKDEVICE(n,p = integer) =
   array [1:n] of array [1:p] of storagecell
   var combuf = command
   path activate; execute; release end
   let D = DISKDEVICE, c = command in
     op D.activate(c) = combuf ← c
     op D.execute = <data transfer by device>
     op D.release = combuf ← nil
end
```

The operation execute represents the action of the disk device executing the command in its command buffer. The details of this action are not relevant here. The path specifies that a program cannot execute release until the device has completed a transfer. The action corresponds to the program detecting that the device is done. Its occurrence in the path is more important than the action it performs. Instead of

setting combuf to **nil**, the body could have been defined as a noop. Its position in the path, however, guarantees that the next command cannot be placed in the command buffer unless the completion of the current data transfer has been detected.

Data transfer requests will be sorted by arrival time per track, i.e. requests for one track are treated FCFS. Grouping requests by track is made possible by

```
type track =
   var com = command(nil)
   path reserve ; val ; leave end
   let t = track, c = command in
      op t.reserve(c) = com ← c
      op t.val = result command; return com
      op t.leave = com ← nil
end
```

The operation leave plays a role similar to that of the operation release in **type** DISKDEVICE. Its usefulness becomes clear in the definition of DISKSCHEDULER (see the definition of access).

The direction in which the head is traveling is represented by a "range type". This is a type definition in which all the constants of that type are listed by name (e.g. **range type** color = red, orange, yellow, green, blue, violet **end**). The operation of **type** direction allows us to change the direction.

```
range type direction = {up,down}
   let d = direction in
      op d.invert = result direction; return if d = up then down else up fi
   end
```

We are now ready for the DISKSCHEDULER. The scheduler keeps track of the first request in both directions in the variables next[down] and next[up]. It changes direction when there are no more requests in that direction. The scheduler makes only one operation available to programs which want to use the DISK device controlled by the scheduler. This operation is "access" and it requires a track number and a command to be executed. The operation access uses the procedures "enter" and "exit" (not available outside the scheduler) in which respectively a request is entered and a next data transfer is scheduled (if any).

```
type DISKSCHEDULER(n,p = integer) =
  array [1:n] of track; var D = DISKDEVICE(n,p)
  var free = array [1:n] of Boolean(true)
  var dir = direction(down), k = integer(0); const Dsize = n
  var next = array direction of integer(0,∞)
  path enter + exit end
  let S = DISKSCHEDULER, i = index, c = command in
    proc S.enter(i) =
      if next[up] = 0 and next[down] = ∞ then  k ← i; D.activate(S[k].val)
      else      free[i] ← false
        if i = k then next[dir.invert] ← i
        else if i < k and next[up] < i then next[up] ← i
            else if i > k and i < next[down] then next[down] ← i fi
      fi fi fi
```

```
 proc S.exit(i) =
 begin var x = integer; if next[dir] = 0 or next[dir] = ∞ then dir ← dir.invert fi
   if 0 < next[dir] < ∞ then
     k ← next[dir]; D.activate(S[k].val)
     if dir = down then
       if some x in [k+1 : Dsize] sat not free[x]
         then free[x] ← true; next[down] ← x else next[down] ← ∞ fi
     else
      if some x in - [1 : k-1] sat not free[x]
         then free[x] ← true; next[up] ← x else next[up] ← 0 fi
   fi fi
 end
 op S.access(i,c) =
    begin S[i].reserve(c); enter(i,c); D.release; S[i].leave; exit(i) end
end
```

The procedure enter immediately activates the requested data transfer if this is the only request in existence. Otherwise, it updates the appropriate next pointer (if necessary). The procedure exit changes the direction if there are no more requests in the present direction. The variables next[up] and next[down] are primarily used for improving the DISK utilization. Without them, procedure exit must search the array of tracks for the next requested track before it can activate the DISK. The variables next[up] and next[down] make it unnecessary that the search through the array precedes the activation of the DISK. This saving of time is important, because, if it is not activated within a critical time limit, the DISK cannot operate at full speed. (The problem can be solved in another way if the array is replaced by two lists, one

for the direction up and one for down. Type track must then be extended with an additional field "num" which records the track number. The bulk of the work is now performed in procedure enter. As before, it immediately activates the requested transfer if this is the first request. Otherwise, it places a new element in the appropriate list (depending on i<k) such that the up-list is sorted by descending track number and the down-list by ascending track number. Procedure exit reduces to a change of direction if the list it is working on is empty and activating the first element of the list in the current direction.)

The search through the array of tracks cannot be omitted, but instead of preceding the DISK activation, it is performed after the DISK has been activated. The construct

some <var> **in** {-}<range> **sat** <Boolean expr>

is equivalent to a logical predicate prefixed by the quantor ∃. (The keyword **sat** reads as "satisfies" or "satisfy".) If the range is empty or the Boolean expression is **false** for all the range values, then the result is **false** and the variable is undefined. If there is a value in the range which satisfies the Boolean expression, the result is **true** and the value of the variable is the leftmost range element for which the Boolean expression is satisfied. The optional minus sign in front of the range means that the range is traversed from right to left. In that case the rightmost range element is returned. (The obvious complement of this predicate is

all <var> **in** {-}<range> **sat** <Boolean expr>

which is short for not **some** <var> in {-}<range> **sat** not <Boolean expr>.)

The programs for enter and exit become substantially shorter if the variables next[up] and next[down] are deleted. Procedure enter reduces to an activation if the

request is the first. Procedure exit amounts to the search for the next and its activation (if one is found). However, it was noted that this may result in poor performance of the DISK device. In this simplified form, the SCHEDULER is essentially the same as the DISK MONITOR presented in [4].

3. CONDITIONALS, PRIORITY AND CONNECTED PATHS

3.1 In some cases the programmer should be able to specify that an operation can be executed only if a certain condition is true. For example, if the **type** stack is defined with the operations push and pop, the former must not be executable when the stack has reached its maximum height and the latter must not be executable when the stack is empty.

A conditional element in a path expression has the form

[<cond.1>:<elem.1>,<cond.2>:<elem.2>,...,<cond.n>:<elem.n>,{elem.(n+1)}]

The conditional element is equal to the leftmost element for which the preceding condition is true. The optional (n+1)st element is the "otherwise". It represents the conditional element if all conditions cond.1,...,cond.n are false.

The conditions in a conditional element are severely restricted. The permitted conditions are Boolean expressions in which the operands are either constants or fieldnames of the type definition in which the path expression is defined. Moreover, all operations which modify the operands of the conditions must occur in the path expression of which the conditional element is a part. These restrictions are necessary to make sure that the evaluation of a condition does not conflict with other

operations on the operands used in that condition. Such a conflict is not possible if the operations which modify operands of a condition occur in the path, because evaluation of the path and execution of one of its elements exclude one another.

Example 4. The operations on a stack are push and pop. The elements of a stack can be of arbitrary type. However, we restrict ourselves to a stack of uniform type, i.e. all the stack elements must be of the same type. When a stack is declared, its maximum height must be specified.

```
type stack(n = integer, t = type) =
   array [1:n] of t
   cons max = n; var top = integer(0)
   path[top = 0: push, top = max: pop, push + pop] end
  let st = stack, x = ref t in
  op st.push(x) = begin top+1; st[top] ← x end
  op st.pop(x) = begin x ← st[top]; top-1 end
end
```

The conditions in this path expression clearly satisfy the restrictions.

3.2. The normal scheduling discipline in a path expression is FCFS. However, there are cases in which execution of two different functions is possible, but the execution of one of these two is more important than execution of the other. If such a fixed priority relation exists between two elements p and q of a selective element p+q, the priority can be indicated in a path by one of the symbols > or <. These symbols have the same precedence as the operator + . E.g.,

```
path f(g > h) k end
```

means that after an execution of f either a g or an h can be executed. However, if an

execution of both g and h is requested, g will be scheduled first.

Stating a fixed priority introduces the problem of a potential starvation. If executions of g are requested so frequently that another request for g has arrived by the time f completes, then h will never be executed. Thus, the priority operator must be applied with care. It can be used in cases in which starvation is not possible or in cases where the starvation is allowed. (An example of the latter is the null operation which is performed on an idling CPU.)

Example 5. The operating system maintains a pool of storage blocks, equal in size, which can be allocated to user programs and will be released in due time. The operations available to a user program are getspace and release. We postulate the existence of a **type** baseaddress, which gives access to a block of storage. The operating system maintains a stack of free blocks. (All free storage blocks are identical, so a stack is as good as a queue.) If there is a state in which either getspace or release can be executed, there is a slight preference for executing release first. This cannot lead to starvation, because there is a finite number of storage blocks in the pool. The number of consecutive executions of release is limited by that number. The type definition for the storage pool is, of course, very similar to the preceding stack example because of our choice to record the free blocks in a stack.

```
decl POOL(n = integer) =
   array [1:n] of baseaddress
   cons max = n; var free = integer(n)
   path[free = 0: release, free = max: getspace, release > getspace] end
   let b = baseaddress in
    op release(b) = begin free+1; POOL[free] ← b end
```

```
    op getspace(b) = begin b ← POOL[free]; free-1 end
end
```

3.3. It is allowed to define more than one path within a type definition. These paths may be independent in the sense that none of the operands in one path occurs in the other, or the paths may share some operands. A multiple path construct can, for instance, be used to express potential parallelism. E.g., the multiple path

```
path p ; r end
path q ; r end
```

specifies that two subsequent p's are separated by an r, that two subsequent q's are separated by an r, and that two subsequent r's are separated by a p and a q. But the multiple path does not specify any ordering between p and q. Therefore, it does not matter in which order p and q are executed in between two subsequent r's. The executions of p and q may even overlap in time. However, the next r cannot be executed until both p and q have been completed.

A path expression allows the execution of only one of the functions named in that path at a time. (In other words, the functions named in a path are automatically embedded in a critical region specific for that path.) The computation of the next state in the path takes place in between two function executions and does not overlap with the execution of one of the functions.

The functions in the multiple path of the preceding example are not necessarily mutually exclusive. In addition, the next state of one path may be evaluated while a non-shared function in the other path is executing. We call such a multiple path structure a "parallel path", because there is an inherent parallelism in the execution of non-shared functions.

A stronger connection between paths is obtained by concatenating several paths into one path. The symbol & is used to represent concatenation. E.g.,

path p ; r & q ; r **end**

is the concatenation of the paths **path** p ; r **end** and **path** q ; r **end**. The concatenated paths are treated as one. This means that only one function named in the path can be executed at a time. In addition, the next state computation takes place in between the execution of functions named in the path, so the next state computation cannot overlap with the execution of one of the functions. All the function executions and the next state computation are mutually exclusive in this case. We call this multiple path structure a "connected path". The given example states that every execution of r must be preceded by an execution of p and an execution of q. The order in which p and q are executed is not specified. However, since p and q occur in a connected path, it is not possible that p and q execute in parallel. Therefore, every execution of r is preceded by either the sequence p;q or by the sequence q;p.

The restrictions imposed upon conditional elements cause no problem in a connected path. In a parallel path, however, the variable operands in a condition can only be modified by operations in the path in which the condition occurs. E.g.,

path [s ≤ t: p,r] & [s < t: q,r] **end**

path [s ≤ t: p,r] **end** and **path** [s < t: q,r] **end**

where p = s ← s - 1, q = t ← t - 1, r = {s ← s + 1; t ← t + 1}. The connected path is correct, because all the operations on the variable operands in the conditions occur in the path. The parallel path violates the restrictions on the variable operands in the conditions, because p in the first path modifies a variable in the condition of the second path and q, in the second path, one in the first path.

Example 6. In the first paper on path expressions [5] we described how a bounded buffer of n slots (n > 1) can be built from the types oneslotbuffer and ringbuffer. The connected paths make it possible to define a ringbuffer which builds directly on the **type** message without having to define an auxiliary **type** oneslotbuffer.

A bounded buffer (or ringbuffer) has a number of N slots which can hold a message (N > 1). The programs which place a message in a slot are called the "senders", the programs which take a message out are called the "receivers". The constraint is that senders and receivers must not operate on a buffer slot at the same time. This can, of course, be achieved by allowing only one sender or one receiver to access the ringbuffer at a time, i.e. by embedding deposit(m=message) and remove(m=message) in one critical region. However, we consider this solution as too restrictive. It is perfectly alright that several senders and several receivers access the ringbuffer at the same time if they access different slots. Thus, the restrictions must be imposed on finding a buffer slot in which a message can be placed or from which a message can be taken.

A buffer slot can be in one of three states: empty, full or inuse. In the state empty, the slot is available for placing a message. In the state full, a message can be taken out. The state inuse indicates that this slot is momentarily not available, because either a message is being placed in this slot, or a message is being taken out.

The **type** ringbuffer makes available two operations on a ringbuffer: deposit(m=message) and remove(m=message). It uses four internal procedures: searchslot, searchmes, addslot and addmes. The procedure searchslot looks for a slot in state empty and the procedure searchmes looks for a slot in state full. The senders should not be able to execute searchslot if all the slots are full. The

receivers should not be able to execute searchmes if all the slots are empty. These constraints will be expressed in a path expression. The procedure addslot is performed by a receiver when it is done taking out a message. A sender performs addmes when it is done placing a message in the ringbuffer.

The search process is slightly improved by the use of two variables d (for deposit) and r (for remove) which respectively point to the last found empty slot and the last found full slot. A search starts at d+1 or r+1 instead of always at the first buffer slot. If the search always starts at the front, the probability of finding a slot in the state we are looking for is smaller at the front than at the end. The variables d and r let a search start at the slots which have been least recently inspected.

```
type ringbuffer(N=integer) =
   array [0 : N-1] of message; cons size = N
   var mesnum = integer(0), slotnum = size, d,r = integer(-1)
   range type slotstate = {empty, inuse, full} end
   var state = array [0 : size-1] of slotstate
   path [mesnum > 0 : searchmes] + addmes & [slotnum > 0 : searchslot] + addslot end
   let rb = ringbuffer, m = message, k = index in
    proc rb.searchmes = result integer
      begin local x = r+1; while state[x] ≠ full do x ← (x+1) % size od
         state[x] ← inuse; mesnum ← mesnum - 1; r ← x; return r
      end
  comment the operator % stands for the remainder function
```

```
    proc rb.searchslot = result integer
      begin local y = d+1; while state[y] ≠ empty do y ← (y+1) % size od
         state[y] ← inuse; slotnum ← slotnum - 1; d ← y; return d
      end
    proc rb.addmes(k) = begin mesnum ← mesnum + 1; state[k] ← full end
    proc rb.addslot(k) =  begin slotnum ← slotnum + 1; state[k] ← empty end
    op rb.deposit(m) = begin local y = searchslot; rb[y] ← m; addmes(y) end
    op rb.remove(m) = begin local x = searchmes; m ← rb[x]; addslot(x) end
end
```

The path expression precludes the execution of deposit if there are no empty slots available and it precludes the execution of remove if there are no messages in the buffer. The path specifies that the search and add operations cannot overlap in time. This guarantees that the elements of the state vector and the variables slotnum and mesnum have a meaningful value. The path does not specify that an execution of searchslot must be followed by an execution of addmes, nor does it require searchmes; addslot. This means that a number of senders and receivers can access the ringbuffer at the same time, but only one at a time can search or add.

This solution differs from the solutions given in [5 and 6] in that here several senders and several receivers can access the buffer, whereas the other solutions allow only one sender and one receiver to access the buffer simultaneously. However, P. Wodon showed that these solutions can be revised to handle several senders and several receivers [7]. The monitor solution given in [4] is very restrictive. It allows only one user at a time, either a sender or a receiver, but not both. It seems not hard to modify the buffer monitor such that it handles the search

and add procedures, but not the buffer operations. Then it also allows several senders and several receivers to access the buffer simultaneously.

Example 7. Another problem that has frequently been discussed is the Readers-Writers problem [8]. A group of "readers" can "read" a data object which they share with a group of "writers" who can "write" the data object. Reading can go on in parallel, but only one writer can write at a time. In addition, writing must not overlap in time with reading.

Since the actions "read" and "write" are of no consequence to the solution of the problem, we will not present a complete type definition for the data objects to be read and written. We confine the solution to the path expressions which restricts the executions of reading and writing.

A writer cannot start as long as reading is going on. It is therefore necessary to distinguish between the states "reading is going on" and its negation. These states and their transitions are easily implemented by counting the number of readers, r. Let read be defined as { rinit ; actual reading ; rquit }, where

procedure rinit = $r \leftarrow r + 1$

and

procedure rquit = $r \leftarrow r - 1$

If r is initialized at zero, the test $r = 0$ reveals whether writing can start or not. This is expressed in the path expression

path [r=0: write, rquit] + rinit **end**

If $r > 0$, writing cannot start, but readers can start and leave. Thus, reading can go on in parallel. If $r = 0$, either a reader or a writer can start. If a writer starts, a reader cannot start until the writer is done.

This solution has a starvation problem. It is possible that the writers will never get a chance if reading is going on. By the time a reader quits, another reader may have performed rinit. If this happens all the time, r will never reach the value zero and writing is impossible.

Writers get a fair chance if no new readers could do rinit after a writer attempts to start. Therefore, we introduce the procedure "writeattempt" and redefine write = {writeattempt ; actual writing}. Reading will die out after a write attempt has succeeded if we add

path rinit + (writeattempt ; write) **end**

to the path above. The additional path does not allow another rinit to start if writeattempt succeeded and r > 0, because the first path does not allow a write to proceed. This means that the element (writeattempt ; write) cannot complete until r = 0.

The first solution favors the readers and the second solution gives both writers and readers a fair chance. The problem discussed most frequently is the one in which the writers have a preferential status. I.e., as soon as a writer attempts to write, no new readers should be able to start reading. The solution of this problem is obtained by a simple modification of the fair solution. The selection operator + in the additional path is replaced by the selection operator < which assigns priority to writing. The path solution is then

path [r = 0: write, rquit] + rinit **end**

path rinit < (writeattempt ; write) **end**

If a reader must wait because writing is going on and another writer arrives later than this reader, the writer is selected when the second path becomes available. Only if

no other writer arrived before the last write operation has been completed, then a reader can perform rinit. The first path assures that writing will not start until all reading has ceased.

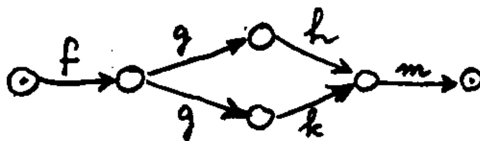
The two paths of the last two solutions form a parallel path instead of a connected path. This means that the non-shared operands rquit and writeattempt can be executed in parallel. It would not make much difference in this case if the paths were connected, because not much is gained by the parallel execution of these two trivial procedures.

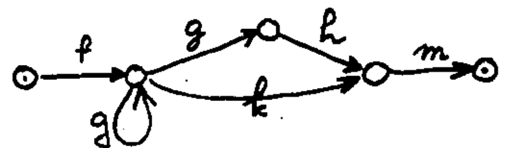
4. PARSING AND IMPLEMENTING PATH EXPRESSIONS

4.1 A path expression has an ambiguity if its graph has two arcs with the same name leaving the same state, but resulting in different states. E.g., the graphs of

path f(gh + gk)m **end** and

path f(gh + g*k)m **end** are





The programmer is allowed to write such ambiguities, because they can be resolved when the path expression is compiled. In the first case the two arcs are replaced by one and the result states are merged into one. This means that the operand g in the path expression is taken out as common factor. The result is

path fg(h + k)m **end**

If the ambiguity involves a repeated element, the trick is to replace this element, $g*$ say, by $\epsilon + gg*$ where ϵ means the empty action. The given example is then transformed into

path f (gh + (ϵ + gg*) k) m **end** =

path f (g (h + g*k) + k) m **end**

The last version is free of ambiguities. (If necessary, g* can be replaced by ϵ + g(ϵ + g(ϵ...+ gg*)...). Applying this rule, **path** f (ggh + g*k) m **end** is transformed into "**path** f (ggh + k + g (ϵ + gg*)) m **end** which reduces to **path** f (k + g (k + g(h + g*k))) m **end**.)

A given path expression can be simplified in a manner similar to the simplification of algebraic expressions. Common factors can be taken out not only from the left but also from the right. E.g.

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

can be written as

path (a+b)p + (a+b)(p+q) **end** = **path** (a+b)(p+p+q) **end**

Since p+p ≡ p, the path can be reduced to

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

It turns out that an unambiguous path expression can be reduced to a canonical form. The proof is essentially the same as the one given for the state reduction of a deterministic finite state machine in [9]. The proof and the algorithms for bringing a given path expression in its canonical form will be discussed in a separate paper.

4.2 The graph of a non-simple path expression has several arcs which carry the same name. Thus, an operand of a non-simple path may be executable in several states.

Let variable STATE indicate for a given path expression p which functions can execute. A function f in this path expression is programmed as

F = p.wait(f); f; p.signal(f)

The signal operation includes the computation of the next state which depends on the current state and the executed function f.

If execution of f is requested and f is not included in the current state, the execution of f is delayed and the process requesting the execution of f is put on a waiting list. It is in principle possible to sort requests per state. However, this means that a requesting process may have to be placed in several waiting lists. If one of the states subsequently allows the execution of the requested function, the process must be removed from several lists. We consider such an implementation as too cumbersome. Instead, a path expression has a single waiting list. If a process P.i requests execution of a function f which cannot be executed right away, a new element (i,f) is added to the list, where i is the process index and f the requested function. The element is appended to the end of the list if no priority is indicated. Otherwise, it is inserted such that the given priority is maintained.

After computing the next state, the signal operation scans the waiting list until it finds an element which can go in this state (if any). If it finds one, this element is removed from the list and the corresponding process is reactivated so that it can execute the requested function.

Only one function named in a path expression can execute at a time. This implies that a new request arriving while one of the functions is executing must be put on the waiting list. If no function is executing, we say that the path is "idle". The operations wait and signal can be programmed using P, V operations on a mutual

exclusion semaphore "mutex" and a set of private semaphores "psem[1:n]", one for each process. The programs are:

```
p.wait(f) =
   begin local x = (i,f)
      P(mutex)
      if idle and x.f ∈ STATE then idle ← false; V(mutex)
      else insert(x into waitinglist); V(mutex); P(psem[x.i]) fi
   end
p.signal(f) =
   begin local x
      P(mutex); STATE ← next(STATE,f)
      if some x in waitinglist sat x.f ∈ STATE then free x; V(psem[x.i])
      else idle ← true fi
      V(mutex)
   end
```

The subpaths of a connected path or a parallel path share one single mutual exclusion semaphore so that only one path is tested at a time. A connected path differs from a parallel path in that the former has one single variable "idle", used by all its subpaths, whereas each subpath k in the latter has its own variable "idle[k]".

In case of a connected path or a parallel path the states of all the subpaths must be tested in which the requested function occurs. The if clause in p.wait(f) is modified for a connected path into

idle **and all** k in [1 : np] **sat** x.f ∈ p[k] → x.f ∈ STATE[k]

where np is the number of subpaths of the connected path. The if-clause in p.wait(f) for a parallel path is

all k **in** [1:np] **sat** x.f ∈ p[k] → (idle[k] **and** x.f ∈ STATE[k])

The modifications in the assignments to the variable idle and in the if-clause in p.signal are self-evident.

The next state is derived from the parse tree which corresponds to the graph of the path expression. In case of a simple path, the next state depends on the executed function. In that case the next state function amounts to copying an attribute of the executed function. Otherwise, if all the operands of a selection are single factors (e.g. (a+b+c)), then the next state only depends on the current state. In that case the next state function amounts to copying an attribute of the current state. If a path expression does not belong to one of these categories, a **case**-statement is attached to each state. The next state is now computed by executing the **case**-statement attached to the current state. The value of the **case**-clause depends on the executed function.

4.3. The use of path expressions does not exclude the possibility of starvation or deadlock. E.g., the parallel path

path a + c **end**

path b + c **end**

may never schedule c if an execution of b is requested while a is being executed and vice-versa. The parallel path

path f p q **end**

path g q p **end**

runs into a deadlock after the first execution of f and g.

Although starvation and deadlock are not impossible, it is easier to detect such problems in a path structure than in programs which use P, V operations on semaphores. In the latter case the problem must be derived from several places in the code. In the introduction path expressions were compared to control structures such as a **while**-statement. Control statements can be misused as much as path expressions. If not programmed on purpose, a starvation problem in a parallel path is comparable to programming an infinite loop, a mistake which can very easily be made.

An additional advantage of path expressions over coding synchronization in line is the detection of deadlocks at compile time. A parallel path can easily be tested for the presence of deadlocks and an error report can be given at compile time. Unfortunately, only deadlocks in path expressions can be detected at compile time. It is still possible to cause deadlocks at run time. E.g. the paths

path f g **end** and

path p q **end**

could be used by two programs P.1 and P.2 such that P.1 successively calls g;p and P.2 q;f. This obviously leads to a deadlock. The best a compiler can do is spot the potential deadlock state. The occurrence of function calls in conditional statements make it impossible to find at compile time which functions will be executed. Thus, path expressions make it easier for a programmer to avoid starvation and deadlock problems, but the responsibility for avoiding these problems is still up to the programmer.

Path expressions make it possible to program the necessary synchronization at a higher level than that of assembly code. Simple path expressions are already powerful tools which would be hard to code in line by P, V operations on semaphores or similar primitive concepts. The examples show that the given rules for writing path expressions are adequate to program useful operating system functions.

A path expression is a regular expression describing the allowable execution sequences of its operands. Several path expressions can be concatenated into one connected path or, by sharing operand names, into a parallel path. A path expression may correspond to an undeterministic finite state machine. The ambiguities can easily be removed by taking out common factors and rewriting repeated elements. The programmer does not have to worry about writing unambiguous path expressions. The ambiguities can be removed by a compiler. The latter also can reduce a path expression to its canonical form.

The testing in a non-simple path expression is slightly more elaborate than in a simple path expression. Connected paths and parallel paths add to the complexity of the test. The programmer must still watch out for unwanted starvation and possible deadlocks. The compiler is able to detect deadlocks present in a connected path or a parallel path. However, the order in which functions, named in a path expression, are called may still cause deadlocks at run time.

The usefulness of path expressions will be demonstrated in the design of an operating system family. Path expressions will be defined as an extension of the process and multiprogramming facilities. At the same time, a modifiable design language is being developed in which path expressions are incorporated. The reduction and compilation of path expressions is incorporated in a compiler for the

design language. More theoretical results about path expressions (and generalization of path expressions) will be presented in E. A. Schneider's thesis by the end of this year.

ACKNOWLEDGMENT

I am grateful for the interesting discussions I had with Roy Campbell while I was in Newcastle, England and while he visited me during the summer of 1974 at CMU. The many hours I spent with Ed Schneider contributed enormously to the development of the path expressions.

REFERENCES

[1] Dijkstra, E. W. "Cooperating Sequential Processes"
In **Programming Languages** (ed. F. Genuys)
Academic Press, New York (1968)

[2] Brinch-Hansen, P. "Structured Multiprogramming"
CACM 15,7 (July 1972)

[3] Lipton, R. On Synchronization Primitive Systems
Thesis, Carnegie-Mellon University (1973)

[4] Hoare, C.A.R. "Monitors: An Operating Structuring Concept"
CACM 17,10 (October 1974)

[5] Campbell, R. H. and Habermann, A. N. "The specification of
Process Synchronization by Path Expressions"
Lecture Notes in Computer Science, Vol. 16
Springer Verlag, Heidelberg, Berlin, New York (1974)

[6] Habermann, A. N. "Synchronization of Communicating Processes"
CACM 15,3 (March 1972)

[7] Wodon, P. Private Communication

[8] Courtois, P. J., Heymans, F. and Parnas, D. L.
"Concurrent Control with Readers and Writers"
CACM 14,10 (October 1971)

[9] Hopcroft, J. E. and Ullman, J. D.
Formal Languages and their Relation to Automata
Addison Wesley, Reading, Mass. (1969)