Indeterminacy, Monitors, and Dataflow*

by

Arvind
Kim P. Gostelow
Wil Plouffe

Technical Report #107

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92717

*This work was supported by NSF Grant MCS76-12460: The UCI Dataflow Architecture Project.

Indeterminacy, Monitors, and Dataflow*

Arvind, Kim P. Gostelow and Wil Plouffe
Department of Information and Computer Science
University of California, Irvine
Irvine, CA 91717

1. Introduction

Operating systems require concurrent or asynchronous programming in order to share computer resources efficiently among competing processes. Programming these systems is further complicated by the fact that processes can interact in a time-dependent manner. Even though this often results in nondeterminism, traditional programming languages have not incorporated nondeterministic operators in a fundamental way. Instead, the usual manner of expressing nondeterminism in such languages is via variables shared among the programs, and over a period of time various programming language extensions have been proposed for structuring this sharing. Still, it is our opinion that traditional languages fall short in dealing with resources in a uniform manner; this should not be surprising since the proposed extensions are not natural to the essentially sequential nature of both the languages and the machines on which these languages execute. If the correct concept of a "resource" is not basic to a language, techniques for sharing a resource must be contrived.

The work described in this paper began with a desire to include some linguistic concept of a resource manager within a dataflow language we have been designing [AGP76]. In doing so, we discovered that dataflow monitors (resource managers) provide a natural way of thinking about resources and especially their scheduling. Dataflow semantics are based upon a program composed of asynchronous operators interconnected by lines along which data tokens (messages) flow, such that when all of the input tokens for a given operator have arrived then that operator may fire (execute) by absorbing the input tokens, computing, and producing an output token as its result. These operations closely match one's intuitive model of resource managers (operators) passing signals (tokens) to one another for the purpose of synchronizing and scheduling resource usage. Previously though, dataflow languages [D73, K73, W75] have dealt only with the expression of highly asynchronous yet determinate computations; however, resource management characteristically involves indeterminate* computation. The introduction here of dataflow monitors and an explicit nondeterministic merge operator for dataflow streams makes dataflow very well suited for expressing interprocess communication and operations on resources.

We also feel that dataflow monitors are superior to monitors found in some conventional languages [BH73, H74] primarily because every aspect of synchronization scheduling that is implicit in other languages becomes explicit in dataflow, and is localized to a specific and easily identified component of the dataflow monitor. Further, the scheduling and enabling of computation is user programmable without resorting to any new primitive function designed only for that purpose. As a result, the correspondence between the structure of a synchronization problem and the structure of the program solution is preserved far better by dataflow than by conventional languages. Also, any indeterminacy in the program is explicit since indeterminacy can be expressed only through the use of a dataflow monitor or a nondeterministic merge operator.

The major points of the paper are illustrated through the use of two examples. The first is a resource manager (implemented as a dataflow monitor) which, with only minor changes, can implement three versions of the readers and writers problem [CHP71, H74]. No other system of which the authors are aware has been able to

*This work was done under the UCI Dataflow Architecture Project and supported by NSF grant MCS 76-12460.

*Indeterminate computation means that the same inputs to a program do not always produce the same outputs; in the case of resource management, one would not expect to receive the same airline seat for every request input to the reservation system.

capture the solutions as easily or clearly. The second example is a solution to a distributed database problem; this example demonstrates the ease with which dataflow languages may be used to solve more complex synchronization problems than are usually used to evaluate the applicability of a new synchronization construct.

2. Introduction to Dataflow and ID

Traditionally, a process is viewed as a completely ordered sequence of instructions (the program) with a single site of activity (represented by the program counter) which moves sequentially through the program's instructions. Branch instructions are needed to modify the normal sequencing. In the context of resource management problems, synchronization primitives explicitly control the relative positions of program counters. In particular, one program counter can not move past a given point until a signal is received from another process, i.e., until another program counter moves past a signal-generating point. This view may be called the "control flow" view of programming since it emphasizes the control aspects of the program. In contrast, "data flow" recognizes only the data dependencies expressed within the program. A program is represented by a set of operators and a partial ordering expressing the data dependencies existing among the operators. Each operator waits for all of its operands to arrive (i.e., to be produced by predecessor operators) and then becomes enabled. Once enabled, the operator may execute at any time and will send its results to other waiting operators. Thus, there may be many sites of activity (operators in execution) within a single program at any instant. The program terminates when no operator is enabled.

This event-driven view of dataflow has analogies in both architecture and operating systems. The design of central processing units of some large scale computer systems (e.g., the arithmetic units of the IBM STRETCH and the IBM 360/91) have been based partially upon the flow of data among instructions. Some operating systems have been based upon the idea of cooperating processes: each process proceeds at its own pace until either a lack of resources or lack of inputs from other processes delays that process (e.g., the RC 4000 [BH73]).

However, program-counter computers are incompatible with a machine language based solely on dataflow semantics; thus it is convenient to hypothesize a dataflow machine. In this machine, data is carried by tokens (packets of bits) that flow through a communication medium and arrive at destination processors. (References [AG77b, DM75] describe two architectures for such machines.) Each processor may execute any enabled operator, and several operators may execute on distinct processors concurrently. Each token carries either an elementary value or a structured value. Elementary values may be integers, reals, strings, booleans, procedure definitions, monitor definitions, monitor names, or errors, while structured values represent vectors with arbitrary selectors [D73]. Variables in dataflow languages do not represent memory locations, but rather the lines (channels) along which tokens flow. A given variable in ID, the Irvine Dataflow language, is either a simple variable or a stream variable. A simple variable carries a single token to each instance of an operator's execution, while a stream variable carries a linearly ordered sequence of tokens to each instance of an operator's execution. Since variables label the lines, no variable may semantically refer to more than one line, and thus no variable may be assigned a value by more than one statement, i.e., a variable must obey the "single-assignment" rule.

ID is expression-oriented with syntax resembling an Algol-like block structure language. All ID programs are expressions. An expression can be a block, a conditional, or a loop expression. The inputs to an expression are the variables referenced but not defined within the expression, and the output is the value(s) computed. An assignment statement is an expression that gives names to the ordered outputs of that expression (similar to, but not equivalent to, an assignment statement in most languages). In this paper, lower-case letters denote simple variables, upper-case letters stream variables.

Examples of ID Syntax

1. a + b is a simple expression; [1,2,3] is a stream of three constants while [] is the empty stream.
2. A block expression that calculates $\sqrt{a^2 + b^2} / \sqrt{a^2 - b^2}$ is:

```
( x ← a*a ;
  y ← b*b
  return sqrt(x+y)/sqrt(x-y) )
```

A block consists of an unordered sequence of statements separated by semicolons and followed by a return clause. If any two statements in a block are interchanged, the meaning of an ID program remains unchanged. Hence, the meaning of

(x ← sqrt(a); y ← x + b/x return x,y)

is the same as

(y ← x + b/x; x ← sqrt(a) return x,y).

A graphical representation of these latter two expressions is given in Figure 1. The inputs to a block consist of all variables referenced in the block which are not defined (not assigned) within the block. The output consists of the value(s) calculated by the return clause.

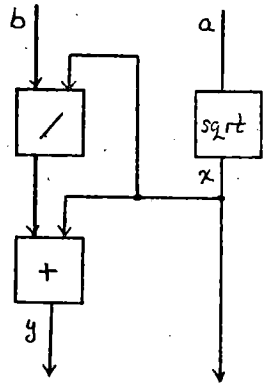

Figure 1

A dataflow expression

The first if statement given above has th
alternative syntax:

```
(if request = "reader"
    then x ← r+1; y ← w
    else x ← r; y ← w+1)
```

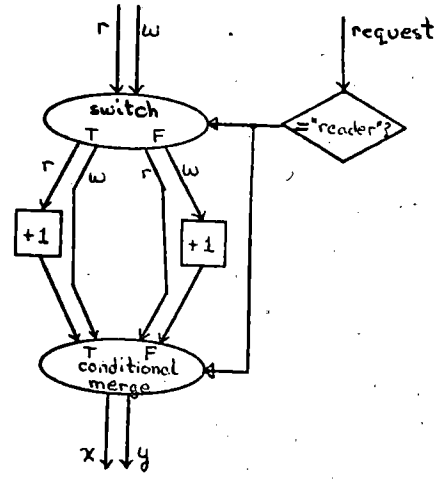

Figure 2

A conditional expression

3. A conditional statement:

```
x, y ← (if request = "reader"
            then r+1, w
            else r, w+1).
```

The inputs to an if expression or statement consist of the union of the inputs to the boolean expression, the then clause, and the else clause. The above if statement requires a token from each of the lines called request, r, and w, and produces two tokens on the lines called x and y. A graphical representation is presented in Figure 2. The switch operator is used to route the tokens on the input lines r and w to the appropriate conditional clause according to the boolean value produced by the predicate (the diamond shaped operator). The conditional merge operator uses the boolean to gather the tokens from the appropriate lines and to output them as the result of the if statement. This requires that the number of outputs from the then clause equals the number from the else clause; thus the following statement is illegal:

```
x, y ← (if request = "reader"
          then r+1
          else r, w+1).
```

4. Structures are created using

```
lt ← <record: i, key: newk>
```

This creates a structure and places that structure as a value on the line lt. The selectors are the strings "record" and "key". The value i may be retrieved using either

```
lt["record"]   or   lt.record
```

5. A loop expression (here we input a stream and output two simple variables):

```
( initial r ← 0; w ← 0
  for each req in REQ do
     r, w ← (if req = "reader"
                then r+1, w
                else r, w+1)
  return r, w )
```

This expression gives the number of readers a writers in the stream REQ by examining in tur each individual token (identified as req) in stream. Here it seems the single-assignment rule is being violated in the body of the loo However, the rule is still valid since the r w on the right-hand side represent those line on which the old accumulated values of the numbers of readers and writers arrive; these

lines are semantically distinct from the lines indicated by the symbols r and w on the left-hand side which represent the output lines of the loop body (see Figure 3 where the r and w from the left-hand side are written as new r and new w). The loop begins execution with a token with value zero on each of the lines initial r and initial w; these tokens are gated through the conditional merge (because of the initial false token represented in the figure by the black dot) and sent to the switch operator. The first token on line req is taken from the stream REQ and, assuming it is not the end-of-stream marker token, req is switched along with the original values of r and w to compute the new values for r and w as new tokens on lines new r and new w. These new values are then sent through the conditional merge (because of the true token generated by the first token req of stream REQ) to the second initiation of the switch operator. The loop continues until the end-of-stream marker token for stream REQ arrives causing a false token to be directed to the switch, whereupon the results are switched out of the loop.

The each operator in Figure 3 does not affect the value of the tokens passing through it. Rather it is needed by the interpreter to change each element of the input stream REQ to a simple value req and to direct each req to a separate initiation of the predicate and switch operators. The interpreter accomplishes this by manipulating control information contained on a token.

Loop expressions emphasize the fact that many simple tokens can appear on a line, yet the underlying interpreter [AG77b] ensures that the proper tokens for a particular initiation of an operation are used. The inputs to the loop expression are the union of all inputs to the right-hand sides of the statements in the initial clause and any externals used within the loop (such as REQ in the example). The outputs are the values computed by the return clause.

Loops also allow for a simplification in the syntax concerning structures. The statement

file[i] ← request

means to append the value of request to the old value of file using the value of i as the selector. This yields a new structure which is the new value of file.

6. The following is an important construct for generating streams in ID:

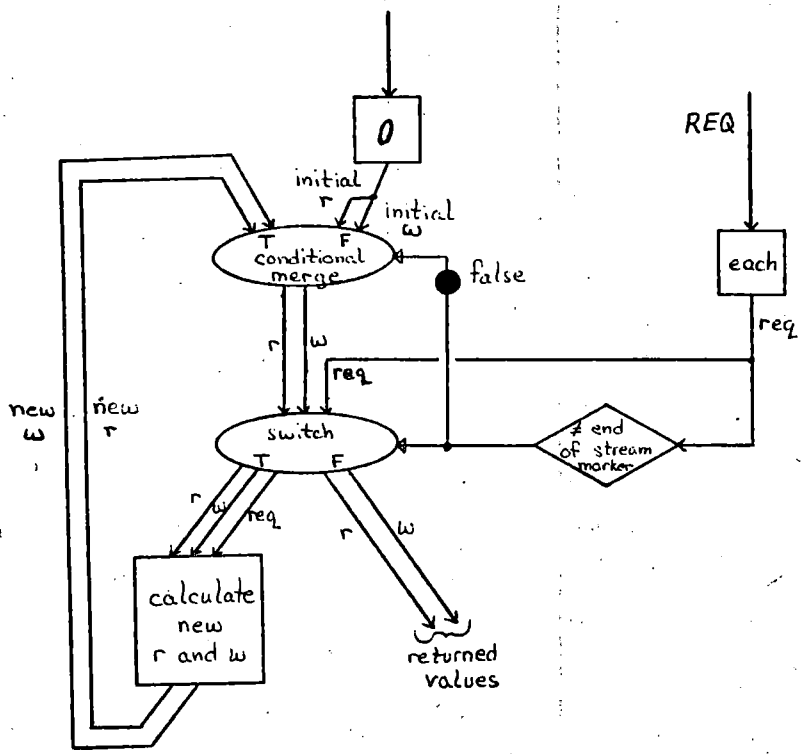

Figure 3

A loop expression

```
( for each a in A; b in B do
      c ← a+b
  return all c )
```

where c is the sum of correspondingly positioned tokens from streams A and B. The all operator performs the inverse function of the each operator. This operator takes a simple token from each iteration of the loop and creates one stream by manipulating the control information contained on the tokens. It does not change the data value carried by the token. The length of the output stream will be the length of the shorter of streams A and B. An abbreviation is:

[each a in A; b in B: a+b]

7. A final example separates the stream of requests REQ into a stream of "readers" and a stream of "writers":

```
( for each req in REQ do
      rq, wq ← (if req = "reader"
                then req, λ else λ, req)
  return all rq but λ, all wq but λ )
```

The phrase "all rq but λ" does exactly what one expects -- a stream of readers is formed by outputting all tokens produced on line rq except those with the value λ. (It is important to realize that in a dataflow language it is impossible to detect the absence of a token. Therefore, we must first produce and then delete it some appropriate time later, that is, after it has been incorporated into a stream.) We also permit the return of all X where X is a stream that is circulated as an entity around a loop each time the loop predicate is true. Then return all X means that the current value of X (an entire stream) is output at each iteration, and the result will be a single stream containing all the individual tokens from each instance of X (as opposed to a stream of streams). For example, if line X successively takes the three stream values

[1,2], [3,4], [5]

the result of all X will be the stream

[1,2,3,4,5].

3. Monitors in ID

The dataflow language described so far can produce only determinate results.* As we said earlier, a nondeterministic operator is essential for ID to be capable of describing the indeterminacy often encountered in operating systems and data base management systems. Resources shared among independent processes are one such source of indeterminacy. In ID, we describe such a resource as a dataflow monitor. A very simple dataflow monitor that implements a file is shown in Figure and its ID description follows:

```
file_def ← monitor (file0)
    (entry REQUEST do
     RESULT ←
        (initial file ← file0
         for each request in REQUEST do
             i ← request.record;
             (if request.type = "read"
                 then file ← file;
                      result ← file[i]
                 else file[i] ← request.data;
                      result ← λ)
         return all result)
     exit RESULT)
```

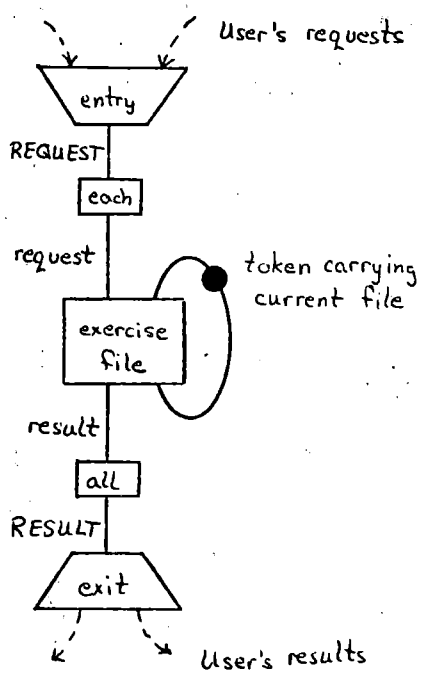

Figure 4

An instance of a file monitor

The variable file0 is a creation-time parameter a is the initial state of the resource; to create instance file_res of monitor file_def with initia state Λ (the null structure or empty file), we wr

file_res ← create (file_def, Λ).

Thus file_res is an instance of file_def, where

*This follows directly from Patil's result stating that any interconnection of a finite number of unconditionally determinate systems is unconditionally determinate [P70, section 3.2.5 of BH73]. It is also shown in [AG77a] that the result computed by a dataflow program is the least fixpoint solution to a set of functional equations. Since a least fixpoint is unique, the result must be determinate.

file0 has the value Λ. The keyword entry in a monitor definition is followed by the monitor input stream identifier, and exit by the result stream identifier. Multiple entrys and exits are defined using several instances of
<entry name>:<stream identifier>. Even though the entry and exit data within a monitor are always streams, an individual use (call) of a monitor supplies only a simple token and expects only a simple result. For example, to use the monitor file_res created above we can write

use (file_res, x)

where x is a simple variable (a single token). The point is that many users are calling upon file_res and each supplies a simple token that states a request to read or write the file. These tokens converge on the single instance file_res and are nondeterministically ordered by the entry operator into the single stream REQUEST. Conversely, the monitor's result stream is broken up by the exit operator into simple tokens that are returned to the individual callers, the ith element of the result stream being returned to the ith caller. (Please note that these are positions in space and not necessarily in time, since the i+1st output may actually be produced before the ith.) The semantics of the entry-exit pair ensure that a result is returned automatically once the request has been processed by the monitor. The identity of the caller is known only to the entry operator which passes it implicitly to the exit operator for forming a return destination address. Hence, the caller must explicitly pass his identity as an input parameter for it to be known inside the monitor.

The monitor file_res guarantees single user access to the resource and enforces the FIFO discipline; note that the entry statement in this monitor is the only indeterminate operator needed. Consider now independent users sharing the resource file_res. A resource manager must satisfy individual requests according to some policy. For the readers and writers problem, the resource manager may permit simultaneous read accesses, but any write access must exclude all other accesses. Figure 5 outlines a resource manager which, with only minor changes, can implement three different scheduling policies corresponding to three different versions of the readers and writers problem. The manager is composed of two logical parts: the agent which performs the actual computation, and the scheduler which blocks or enables individual requests within the agent. We emphasize the word "logical", in that the scheduler possesses no new primitive functions in order to carry out its work, and is entirely user programmable.

For the resource manager we are now describing, each request enters the queue READQ or the queue WRITEQ according to the type of the request (i.e., which named entry port was used). Each queued request will match with an enabling signal from the streams READ_ENABLE or WRITE_ENABLE that are generated by the scheduler which then allow queued values to be released to the access_resource routine. This is done using a when clause, where an expression followed by "when t" specifies that evaluation of the expression must be delayed until t arrives (this is easily accomplished in dataflow). Proper operation of the resource manager requires that the scheduler be notified whenever (1) a request enters the monitor or (2) a request completes its read or write access. Since these signals are nondeterministically generated, we merge them within the resource manager to form a single stream X of signals to the scheduler. Thus, nondeterminacy may appear in two ways in ID: in an entry statement, and in a merge statement. One difference is that entry expects simple inputs and produces a stream as output, while merge expects streams as input and produces a stream as output.

In the programmed solution of the resource manager given below, the scheduler state is represented by the number of active readers (ra), the number of active writers (wa), the number of

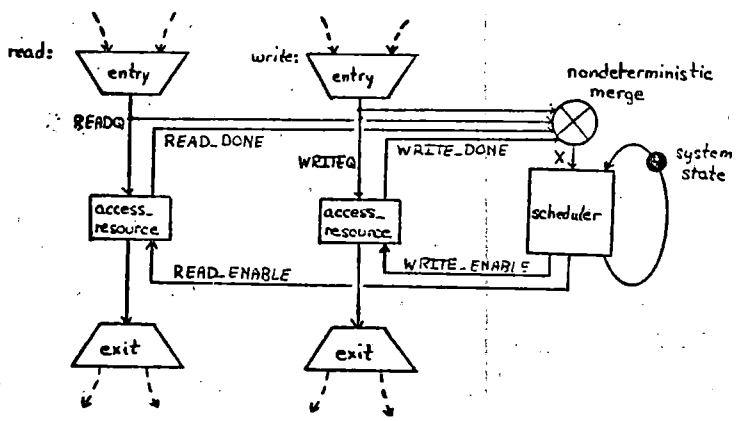

Figure 5
A resource manager

waiting readers (rw), and the number of waiting writers (ww). The scheduler enables requests to leave the waiting queue by producing a stream of reader enabling tokens (RE) or one writer enabling token (we). Note that

1) wa<=1 at all times,
2) if wa=1 then ra=0, and
3) if ra>0 then wa=0.

Version 1: (Hoare [H74]) A new reader is not permitted to proceed if a writer is waiting, and a readers that are waiting when a writer completes a allowed to proceed. This scheme prevents indefinit exclusion ("starvation") of both the readers and th writers. The program for this version of the problem is:

```
resource_manager ←
 monitor (file)        ! any file monitor resource such as file_res above !
 (entry read: READQ;
       write: WRITEQ do

    ! this is the agent code for a read request !
    READ_RESULT,READ_DONE ← [each r in READQ; re in READ_ENABLE:
                             (s ← access_resource(file,r) when re
                              return s, "read exit" when s)];

    ! this is the agent code for a write request !
    WRITE_RESULT,WRITE_DONE ←[each r in WRITEQ; we in WRITE_ENABLE:
                              (s ← access_resource(file,r) when we
                               return s, "write exit" when s)];

    X ← merge(READQ,WRITEQ,READ_DONE,WRITE_DONE);


    ! the scheduler begins here --
         its function is to produce enabling signals !
    ! the input is stream X, the outputs are streams
         READ_ENABLE and WRITE_ENABLE !

    READ_ENABLE,WRITE_ENABLE ←
        (initial rw, ww, ra, wa ← 0,0,0,0    ! initial state !
         for each x in X do
             rw, ww, ra, wa, RE, we ←        (A)
              (case
                 x="reader" => (if wa=0 and ww=0
                                   then rw, ww, ra+1, wa, ["go"], λ
                                   else rw+1, ww, ra, wa, [], λ)
                 x="writer" => (if wa=0 and ra=0
                                   then rw, ww, ra, 1, [], "go"
                                   else rw, ww+1, ra, wa, [], λ)
                 x="read exit" => (if ra=1 and ww>0
                                      then rw, ww-1, 0, 1, [], "go"
                                      else rw, ww, ra-1, wa, [], λ)      (D)
                 x="write exit" => (if rw>0 -> (B)
                                      then 0, ww, rw, 0, (for i from 1 to rw do
                                                          return all "go"), λ
                                      else (if ww>0 -> (C)
                                               then rw, ww-1, ra, wa, [], "go"
                                               else rw, ww, ra, 0, [], λ))) (E)
        return all RE, all we but λ)

    ! the scheduler ends here !

    exit read: READ_RESULT;
         write: WRITE_RESULT          ! end of the monitor !)
```

To embed the shared file file_res within the resource manager, we pass file_res to it at creation time:

file_manager←create(resource_manager, file_res)

Requests to read file_res can now be performed by writing

use (file_manager.read, request)

Write requests are handled in a similar manner.

Version 2: (Problem 1 of [CHP71]) No reader is kept waiting unless a writer has already acquired the resource. Starvation of writers is possible. This means that the condition for generating an enabling signal for a "reader" is relaxed from wa=0 and ra=0 to simply wa=0. This is accomplished by deleting the code marked (A) from the first case condition in the scheduler of Version 1.

Version 3: (Problem 2 of [CHP71]) No reader is allowed to proceed if a writer is waiting. Starvation of readers is possible. This does not affect the scheduling in the case for a "read exit" because the same condition applied in Version 1. However, for a "write exit" the scheduler must check for a waiting writer (ww>0) before testing for waiting readers (rw>0). In particular, the code for Version 1 in position (B) is interchanged with that in position (C), and the code in (D) is interchanged with that in (E).

This example illustrates an advantage of dataflow. The program presents explicitly the essential components of the problem: the agent with separate reader and writer queues, and the scheduler which clearly shows the conditions under which enabling signals are sent and which thus is easily changed to implement different policies. In all the problems we have programmed, our experience has been that the scheduler policy is explicit and easily altered to suit various scheduling criteria.

(Again, please note that the distinction made between agent and scheduler is only for emphasis, and that no special scheduling primitives are required.) Also, the scheme is modular, as illustrated here by the embedding of the file resource monitor within the resource manager monitor.

4. A Problem in Distributed Databases

Consider an airline reservation system which is to be built from many small databases, each controlled by its own monitor. To hide the partitioning of the database from users, each user will send requests to an "interface" monitor which will route the requests to the appropriate database. Ideally, most of the requests an interface monitor receives will pertain to local data. For simplicity we shall assume that each local database is a single file comprising many records. (See Figure 6.)

There are only two types of requests that can be made:

1. Read (fetch) record i from file f and lock it, so that the record is marked as missing. The request format is

   <type: "lock", file:f, record: i>.

   In response, the system returns a copy of the fetched data record i and a unique key k (a random number used to later unlock the record) of the form

   <record: i, key: k>.

2. Unlock record i from file f with key k, and update it with data d. The request format is

   <type: "unlock", file: f, record: i, data: d, key: k>.

   There are two possible responses: if the correct key has been supplied and the record was previously locked in the file, the response is "successful"; otherwise the request is ignored and the response is "unsuccessful".

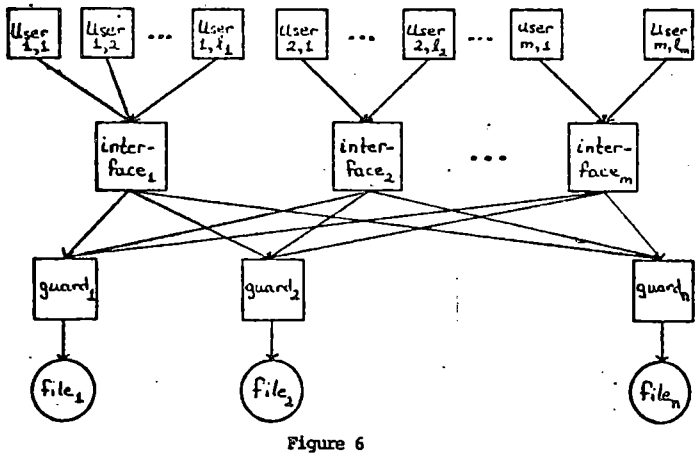

Figure 6

Overview of the Distributed database system

We shall require each user to make requests in packets; in particular, all lock requests must be made in one packet. However, this is not sufficient to avoid deadlock, because interface monitors can still send lock requests in an order that results in a circular wait. The statement of the problem precludes a central scheduler to resolve these conflicts, so we shall use the "ordered resource usage" method of deadlock avoidance. This method requires the interface monitor to make lock requests to the files in some fixed system-wide order and not to make a request to a second file until the requests to the first file have been satisfied. We assume lock requests are made by

use (file[f].lock, group) when t

where group is the packet of lock requests for file f and file[f] is the resource manager monitor for the file. The variable t is the condition of having received the response to any previous packet of requests (i.e., to file[f´] where f´<f). The result will be a packet of responses of the type <record: i, key: k> mentioned above.

To achieve greater internal asynchrony, a resource manager monitor changes the packet of lock requests from an array into a stream of lock requests LOCKQ, thus making each request independent of the other requests in the packet. The reverse takes places before leaving the monitor. Each lock request in LOCKQ waits for an enabling signal in LOCK_ENABLE from the scheduler in a manner very similar to that used in the readers and writers monitor of Section 3. However, the relocking of a locked record must be postponed without interfering with lock requests for other records. To accomplish this we form a separate queue for each record. The unbounded and varying number of records in a file dictates that these queues be created dynamically. Since a queue is actually just a dataflow stream, this can be done easily using an ID construct for creating and manipulating dynamic streams.

The ID construct

dswitch I ← S via L

creates a distinct logical stream I_j for each distinct value j in stream L. Stream I_j will contain only those tokens of S for which the corresponding token of L has the value j. Thus

dswitch I ← [a,b,c,d,e,f,g]
via [1,3,2,3,3,1,1]

would create the three logical streams

I_1 = [a,f,g], I_2 = [c], I_3 = [b,d,e]

The ID construct

dmerge I via M

is the inverse of dswitch. Thus demerge creates a single stream using the dynamic stream I according to the specification stream M by removing one token from the logical stream I_j when the next token from stream M carries the value j. Using the example above,

dmerge I via [2,3,3,1,3,1,1]

would return the stream [c,b,d,a,e,f,g]. Note that both dswitch and dmerge are deterministic operators.

To create a dynamically varying number of queues, one queue for the lock requests for each record, we would write

```
REQUEST_RECORD ← [each req in LOCKQ: req.record
dswitch DREQ ← LOCKQ via REQUEST_RECORD
```

Also, to direct the enabling signals (which have form <record: i, key: k>) to the correct logical stream we would write

```
KEYS ← [each signal in LOCK_ENABLE: signal.key]
SIG_RECORD ← [each signal in LOCK_ENABLE:
                signal.record];
dswitch DKEY ← KEYS via SIG_RECORD
```

Finally, any expression enclosed by dswitch ... dmerge is logically created and independently acted upon by each set of dynamic streams (please see Figure 7). Thus, the final code for queueing lock requests is

```
REQUEST_RECORD ← [each req in LOCKQ: req.record
KEYS ← [each signal in LOCK_ENABLE: signal.key]
SIG_RECORD ← [each signal in LOCK_ENABLE:
                signal.record];
RESULT ← (dswitch DREQ ← LOCKQ via REQUEST_RECOR
                DKEY ← KEYS via SIG_RECORD do
          DRESULT ← [each req in DREQ; key in DK

                                <record: use (file,rec
                                          key:key>)
          dmerge DRESULT via REQUEST_RECORD)
```

Note that the above code allows each record to be treated independently of all other records. Now we describe exactly how the scheduler for the resource manager monitor operates.

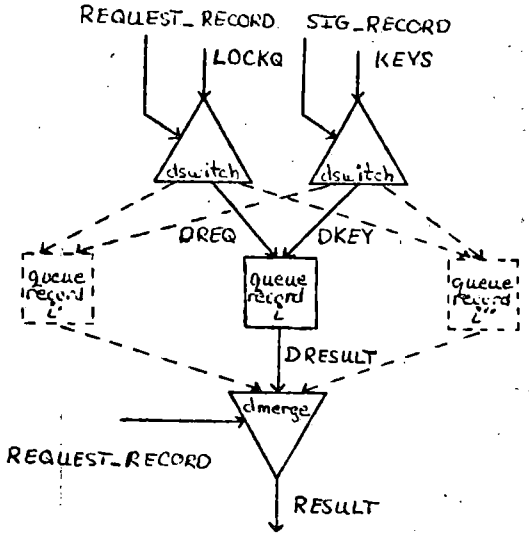

Figure 7

Dynamic streams

The scheduler keeps track of all the records that are locked (lock[i] = "yes" if i is locked), the keys of the locked records (k[i] is the required key if i is locked), and the count of lock requests that are waiting on record i (the value of w[i]). Since the scheduler must be informed after an unlock operation has been completed, a DONE stream is generated from the unlock operation. The scheduler code that generates a lock or unlock enabling signal follows*:

```
X ← merge(LOCKQ,UNLOCKQ,DONE);
LOCK_ENABLE,UNLOCK_ENABLE ←
    (initial lock,k,w ← Λ,Λ,Λ; newk ← random(seed)
     for each x in X do
         i ← x.record; type ← x.type;
         (if type="unlocked"
             then ! x is a token from the DONE stream !
                 (if w[i]=Λ then lt,ut← λ,λ; lock[i]← Λ
                           else lt,ut ← <record: i, key: newk>,λ;
                                newk ← random(newk);
                                k[i] ← newk;
                                w[i] ← (if w[i]=1 then Λ else w[i]-1))
             else ! x is a new request !
                 (case
                     type="lock" and lock[i]="yes" => lt,ut←λ,λ;
                                                      w[i] ← (if w[i]=Λ then 1 else w[i]+1);
                     type="lock" and lock[i]≠"yes" => lt,ut←<record: i, key: newk>,λ;
                                                      lock[i]← "yes";
                                                      newk← random(newk);
                                                      k[i]← newk;
                     type="unlock" and lock[i]="yes" and x.key=k[i]=> lt,ut←λ,"successful";
                                                                      k[i]← Λ;
                     else lt,ut← λ,"unsuccessful"))
    return all lt but λ, all ut but λ)
```

5. Conclusions

A great deal of theoretical research has been done in the last fifteen years that deals with parallel program schemes [KM69] graph models (e.g., [R69]), and Petri nets [P62]. Dataflow -- the notion that operations should be data driven -- has also found periodic favor in various system designs. Perhaps the time has arrived for dataflow (which is a practical extension of the theoretical research) to be taken as a proper linguistic model for programming continuously operating systems. We have tried to demonstrate in this paper that dataflow monitors (resource managers) provide a natural way of thinking about resources and especially about the scheduling of resources. Previously, dataflow languages have dealt with the highly concurrent or asynchronous expression of determinate computations. However the notion of streams together with a nondeterministic merge operator makes dataflow very well suited for expressing interprocess communication and operations on resources. A dataflow monitor (that is, an entry-exit pair) further provides a good linguistic feature to encase a resource and all the programs associated with the resource within a single programmed unit.

Dataflow monitors are superior to monitors in conventional languages because the scheduling policy may be programmed explicitly in a dataflow monitor. For example, assume process p_1 is waiting on some condition c_1 and process p_2 is waiting on conditions c_1 and c_2. Further, assume condition c_2 has been true when condition c_1 becomes true. A monitor in a conventional language can not be programmed easily to give priority to one of the two waiting processes. This is because the priority would be intrinsic to the routine that manipulates the queues associated with each condition. This routine is generally not treated as part of the monitor definition and hence is not programmable. As we have shown using the two examples in this paper, all such scheduling decisions are an integral part of a dataflow monitor definition. It is largely because of this explicit scheduling that the structure of the readers and writers problem is reflected so neatly in dataflow.

Another very important fact is that indeterminacy in dataflow programs cannot "sneak in through the back door" as it sometimes does in

*Note that the conditional clauses utilize a syntactic shorthand within the loop and contain assignments to only the variables of lock, newk, k, and w when and where they are actually to change values.

sequential languages (a reader who has programmed interrupt handlers will appreciate this). In dataflow, a nondeterministic merge operator or a monitor call must be specified in a program precisely where the programmer expects time-dependent behavior from his program.

Finally, we do not believe that dataflow should be limited to describing just those systems where understanding the computational process is at least as important as the results being produced (i.e., operating systems). Rather, we believe that dataflow is a superior vehicle for almost any programming problem.

Acknowledgements

Our early interest in dataflow was due to the fundamental work of Jack Dennis. We are grateful to Peter Denning for his useful comments, and to Shirley Rasmussen for typing many of the early drafts.

References

[AG77a] Arvind and Gostelow, K.P. Some relationships between asynchronous interpreters of a data flow language. Preprints of IFIP Working Conf. on Formal Description of Programming Concepts, Vol. 1, North-Holland, New York, 1977, pp. 4.1-4.25.

[AG77b] Arvind and Gostelow, K.P. A computer capable of exchanging processing elements for time. Information Processing 77, B. Gilchrist (Ed.), North-Holland, New York, 1977, pp. 849-853.

[AGP76] Arvind, Gostelow, K.P., and Plouffe, W.E. Programming in a viable data flow language. Technical Report 89, Dept. of Information & Computer Science, Univ. of California, Irvine, CA, Aug. 1976.

[BH73] Brinch Hansen, P. Operating Systems Principles. Prentice-Hall, Englewood Cliffs, NJ, 1973.

[CHP71] Courtois, P.J., Heymans, F., and Parnas, D.L. Concurrent control with "readers" and "writers". Comm. ACM 14, 10 (Oct. 1971), 667-668.

[D73] Dennis, J.B. First version of a data fl procedure language. Computation Structures Group Memo 92, Lab. for Computer Science, Cambridge, MA, Nov. 1973. (Revised in Aug. 1974; reissued as Technical Memorandum 61, May 1975.)

[DM75] Dennis, J.B. and Misunas, D.P. A preliminary architecture for a basic data-flow processor. Proc. Second Annua Symp. on Computer Architecture, Jan. 197 pp. 126-132.

[H74] Hoare, C.A.R. Monitors: An operating system structuring concept. Comm. ACM 17 10 (Oct. 1974), 549-557.

[KM69] Karp, R.M. and Miller, R.E. Parallel program schemata. J. Comptr. Sys. Sci. 3 2 (May 1969), 147-195.

[K73] Kosinski, P.R. A data flow language for operating systems programming. SIGPLAN Notices (ACM) 8, 9 (Sept. 1973), 89-94.

[P70] Patil, S.S. Closure properties of interconnections of determinate systems. Record of the Project MAC Conf. on Concurrent Systems and Parallel Computation, June 1970, pp. 107-116.

[P62] Petri, C.A. Kommunikation mit automaten. Institut fur Angewandte Mathematik der Universitat Bonn, Wegelerstrasse 10, Bonn, 1962.

[R69] Rodriguez, J.E. A graph model for parallel computation. MAC-TR-64, Project MAC, M.I.T., Cambridge, MA, Sept. 1969.

[W75] Weng, K.-S. Stream-oriented computation in recursive data flow schemas. Lab. for Computer Science, Cambridge, MA, Oct. 1975.