Proceedings of the 3rd International Conference on Software Engineering 1978, pp72-81.

Information Systems: Modelling, Sequencing and Transformation

M A Jackson

Michael Jackson Systems Limited
101 Hamilton Terrace
London NW8 9QY England

Keywords: function, hierarchy, model, network, optimisation, procedure, process, program transformation.

Abstract: Specification and design of an information system conventionally starts from consideration of the system *function*. This paper argues that consideration may more properly be given first to the system as a *model* of the reality with which it is concerned, the function being subsequently superimposed on the model.

The form of model proposed is a network of sequential processes communicating by serial data streams. Such a model permits a clear representation of change or activity over time, and it also prevents over-specification of sequencing by separating problem-oriented from solution-oriented sequencing constraints. The model, however, cannot be efficiently executed on uniprocessor hardware without transformation. Some relevant kinds of transformation are mentioned, and the derivation, by means of them, of conventional information system configurations from the proposed model.

1. INTRODUCTION

The design of an information system may be approached from a *functional* point of view. From the original question 'what is the system for?' we progress naturally to asking 'what should the system do?', or 'what should be the system's *function*?'. Since the system's function is usually very complicated, we seek a method of mastering its complexity, and we find that method in step-wise refinement, or in its homely country cousin, top-down design: a first, gross, statement of the system function is elaborated or refined in successive stages until a hierarchy of functions, subfunctions and sub-subfunctions has been constructed. The fundamental structuring device is the procedure invocation, and the design task ends when the lowest level procedures of the system hierarchy can be implemented by the elementary operations of the available hardware/software machine.

The functional view is readily extended from system design back to requirements analysis, and is widely accepted as the implicitly necessary basis for all thinking about systems development methodology (1). I would wish to argue that this functional view, even allowing for the considerable degree of over-simplification and perhaps caricature in the above description, has severe disadvantages, and that an alternative approach merits serious consideration.

1.1 Disadvantages of the functional view

The first disadvantage is that functional design is simply very difficult for all but the most trivial problems and the most inspired designers. By hypothesis, the total function is too complex to be grasped in every detail by a single mind at a single time: how then can the designer, as he makes his first refinement step, have confidence that unforeseen details will not invalidate his decisions? How can he decide which decisions to take now and which to postpone? Consider, for example, the small problem of printing out, in order, the prime numbers up to 1000. Should the first refinement be:

```
begin generate table of primes;
      print table of primes
end
```

or:

```
for n:=2 step 1 until 1000
  if n is prime then print n fi
```

or:

```
begin n:=2;
  while n <= 1000
    do print n;
       generate next prime n
    od
end
```

or perhaps something altogether different? Only a prior acquaintance with the problem, or great foresight, or intuition, or good fortune, can guide the designer to the best decision. At the outset of the design activity, when the problem is not yet solved, the set of apparently possible solutions by functional refinement is too large, and there are few, if any, effective criteria for choice.

The second disadvantage is that system function is subject to almost arbitrary change: as soon as the system is built, or earlier if we permit it, the customer is knocking on our door with requests for specification changes. Distressingly often an apparently small change in specification causes a large disruption in the system structure: it is not easy to anticipate these specification changes nor to design systems which are robust enough to withstand them. We need, at least, to be able to categorise all possible specification changes according to their expected cost; we need to be able to say to the customer 'these changes will be very expensive, those will be moderately expensive, and those will be cheap'. It is hard to see how this can be done on the basis of a purely functional design.

The third disadvantage appears at first in the guise of a benefit: there is a conceptual unification of the design process from the highest to the lowest level of the system. We may consider each level of the system to be, conceptually, a machine; at the lowest level we have the hardware machine providing elementary executable operations such as add and subtract, multiply and divide; at the next level up we have a software machine built of simple procedures providing such functions as sin, square root, arctan; at the highest level we have a software machine providing the operation 'solve this problem'. Although we may choose, at different stages, to engage in bottom-up or in top-down design, to raise the machine platform upwards towards the problem or to lower the problem level downwards towards the machine, the product of the design activity is the same for both directions: a hierarchy of functional procedures. The crucial disadvantage of this unification is that there is no clear point at which we lay down the designer's tools and take up those of the constructor, at which we abandon the pencil and circuit diagram and take up the wire-wrap gun. The software engineering practitioner always works with pencil and paper—or, if he is in fashion, with screen and keyboard. As a result, potentially valuable distinctions become blurred; decisions that properly belong to a late stage of construction are taken, perhaps unconsciously, in an early stage of design. This is particularly true of decisions about sequencing: the problem of scheduling a multiprocess system on a uniprocessing machine is often solved prematurely, before the processes to be scheduled have been fully understood.

1.2 Modelling

I suggest that these difficulties may be lessened by demoting the idea of 'function' to a secondary role. Instead of viewing the system as primarily a device for doing something, for computing its outputs from its inputs, we should view it primarily as a model of the reality with which it is concerned. The 'function' of the system is secondary, and is considered to be *superimposed* on the underlying model: a given model can support many functions. Of course, the choice of model must ultimately depend on the functions which the system is to provide, but it does not depend on them in detail; rather, it depends on the customer's view of his world, for it is that which we must model. Modelling has inevitably played an implicit part in all information system design. In simulation it plays an explicit part, and we sometimes find the separation of model from function also appears in a fully explicit form. For example, after describing a job-shop model, Dahl and Hoare (2) write:

> "The model above should be augmented by mechanisms for observing its performance. We may for instance very easily include a 'reporter' process, which will operate in 'parallel' with the model components and give output of relevant state information at regular simulated time intervals."

The following programming problem is a small illustration.

"An organisation provides resources for use by the public. (It might be, for example, that the resource is a time-sharing system, or boats on a pleasure lake). Each session of use by a customer has a starting time and an ending time, which are recorded, in chronological order, on a magnetic tape file. Each record of the file contains the session-identifier, a type code ('S' for start and 'E' for end) and the time; the file contains a 'S' and an 'E' record, in that order, for each session in one day.

"It is desired to produce a report from the tape showing the total number of sessions of use and the average session time. Time for a session is computed as Te–Ts, where Ts and Te are the start and end times respectively for the session."

By considering the function of the program we arrive, with little difficulty, at the following text:

```
P: begin sessions :=0; totaltime:=0;
         open tape; read tape;
         while not eof(tape)
               do if code = 'S'
                     then begin
                          sessions:=sessions+1;
                          totaltime:=totaltime-Ts
                          end
                  else totaltime:=totaltime+Te
                  fi
                  read tape
               od
         print 'number of sessions = ', sessions;
         if sessions not = 0
            then print 'average session time = ',
                       (totaltime/sessions)
         fi
         close tape
   end
```

This program certainly solves the problem in the sense that it satisfies the 'functional requirement'. But it is not based on any explicit model of reality, and the implicit model, so far as we can discern it, is grossly inadequate. In the specification, informal as it was, a reality was described in which there were sessions of use of the organisation's resource, each session having an individual existence. These individual sessions should have been modelled in the program but have not been: there is nothing in the program which can be identified with an individual session.

The penalty for this failure in modelling is that the program is entirely incapable of modification to meet changed requirements. For example, if we are required to print out the longest session time of the day, or to split the report into two parts, one for sessions starting in the morning and one for sessions starting in the afternoon, or to accommodate an imperfect tape which may contain unreadable records—if we are faced with any of these requests we can only abandon the program and write a new one.

I am suggesting that modelling must not only be more explicit, but must be given a primary position in system design activity. The designer should start, not by considering the required function (what are the inputs and outputs of the system?) but by considering the reality to be modelled (what is a session? what is a customer?). The model is then designed and built in a basic form which is in principle, though not necessarily in practice, executable. The action of the model then reflects the action of the real world: as each session is a recognisable entity in the real world so it is in the model; as each session has a beginning and ending in the real world, so it does in the model also. Then, and only then, the function of the system may be superimposed on the model: just as we might employ a human observer with a stopwatch to collect information from the real world, so we can employ a model observer to collect information from the model.

The functions that can be superimposed are, evidently, limited by the model: if the system does not model the individual customers, then we cannot obtain information about customers; if the system does not model the individual parts of the resource (for example, the individual boats on the pleasure lake), then we cannot obtain information about those. How, then, is it easier to deal with specification

changes? It is easier because the model is explicit, and defines, as it were, the convex hull of all the functions it can possibly support. With the explicit model we can go to the system user and define the requirements which we can meet, now and in the future, in an easily intelligible way—we can say 'you will be able to get any information you want about sessions, but you will not be able to ask questions about individual customers or individual boats'. The set of functions thus described is readily formulated and readily understood; its simplicity is due to the fact that it is unrelated to considerations of system implementation.

The basic form of model is a network of processes, one process for each independently active entity in the real world. These processes communicate by writing and reading serial data streams or files, each data stream connecting exactly two processes; there is no process synchronisation other than by these data streams. Implementation of such a model on currently available uniprocessor or pauciprocessor machines, requires substantial transformation of the program texts.

In the body of this paper I would like first to describe the basic model in more detail, then to give some small examples of models and superimposed functions, and finally to discuss some transformations by which systems of a more conventional appearance are derivable from the basic model. Related transformations and equivalences among different conventional systems are discussed by Dwyer (3).

2. THE BASIC MODEL FORM

2.1 Entities

Each independently active entity in the real world is represented in the model by a process. Thus, if in the real world there are customers, orders, suppliers, parts and employees, then in the model there must be a process for each customer, for each order, for each supplier, for each part and for each employee. As the entity progresses through its lifetime in the real world, so the modelling process progresses through its program text; we may, in principle, follow the lifetime of the entity by following the text pointer of its program. If an order has been placed, but the ordered parts have not yet been allocated, then the text pointer of the order process will have a value lying between the part of the program text which models the placing of the order and the part which models the allocation of the parts.

A process may have local variables, which can represent attributes of the entity (for example, a customer's address) or summaries of its history (for example, a customer's current balance). A process is considered to be executing on a dedicated processor; there is therefore no notion of process scheduling or sharing of processors.

2.2 Process communication

Process communication is entirely effected by serial data streams; a process may have any number of input and output data streams, but each data stream is written by only one process and read by only one process. Each data stream consists of a sequence of records, each record being the object of a 'write' and a 'read' operation in the producing and consuming process respectively. The data stream itself is considered to be an infinitely expansible queue; the consuming process is blocked when it attempts to execute a read operation on an empty queue, but the producing process is never blocked by a write operation. The write operation models an action by the producing process which affects the consuming process; the read operation models the readiness of the consuming process to be appropriately affected. Kahn and MacQueen (4) and Hoare (5) discuss some of the implications of such a scheme.

2.3 System inputs and outputs

In addition to data streams which connect two processes, there are data streams which connect the system to the real world. In a pure model, with no superimposed functions, there are no system outputs: the data streams connecting the system to the real world are therefore written outside the system and read by processes within the system. The meaning of the records of these data streams is

somewhat different from the meaning of the records of other data streams: they provide synchronisation and co-ordination of the model with the real world. Thus, for example, if the customer in the real world places an order, the customer process in the model must be stimulated to do the same; the data stream input which achieves this is not therefore an action affecting the customer, but rather an action to be performed by the customer himself.

Superimposed functions will, in general, produce outputs. These may be obtained either by suitable output operations embedded within the model processes themselves, or by the addition of reporting processes which are privileged to examine the local state variables of the model processes and to report on their values. We may refer to functions implemented in these two ways as *embedded* functions and *imposed* functions respectively.

2.4 Process classes

Each process may be considered to belong to a process *class* (6). For example, there may be a class of customer processes, a class of order processes, and so on. The class of a process determines its program text, and also determines how the process fits into the system network. Each data stream too may be considered to belong to a data stream class, which determines the possible sequences of records in the data stream: with each data stream class a grammar is associated, and each data stream of the class is a sentence in that grammar. More properly there are two grammars associated with each data stream class, one relevant to the producing process and the other relevant to the consuming process; a data stream of that class is a sentence in both of these grammars. The structure of the program text for a process class is based on the structures (ie grammars) of the data streams which it reads and writes (7).

A process may create a process of another class and communicate with it by data streams. Having created a process, it may subsequently terminate it, by executing a 'close' operation on the data stream read by the created process. A process which is created and terminated within the system in this way may be called a *child* process of its creator. Those process which are not children of other processes in the system must be created and terminated from outside the system, via the external data streams which they read from the outside world. The ideas of set membership and set ownership in database systems may be compared (8).

3. SOME SMALL EXAMPLES

We may illustrate these rather informal ideas about modelling by some small examples.

Example 1

The system to be modelled consists only of a set of inert and unchanging objects: it is a collection of vehicle license numbers made by a young boy for no purpose beyond the simple joys of collecting.

The model is a set of degenerate processes, one for each license number in the collection. The local variables (actually, constants) of each process are the license number itself, the date it was spotted by the collector, and the make of vehicle. There are no data streams in the system and no connections among the degenerate processes. The program texts consist of no more than the declarations of the local data, there being no executable part.

Clearly, only imposed functions are possible in a system of this kind: embedded functions would never be activated. Since the model itself is merely a set of inert objects, with no interconnections, an imposed function may treat the collection as having any desired structure: the responsibility of traversing the set of license numbers according to the requirements of the desired structure lies with the imposed function, not with the model system.

Example 2

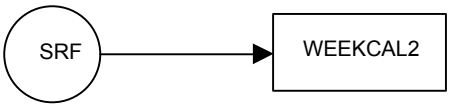
The system is a simple calendar in which the day of the week varies over an indefinite period which begins on a Sunday.

The whole system consists of a single process:

```
WEEKCAL: begin
```

```
        while true do
                  day:=Sunday;
                  day:=Monday;
                  ...
                  day:=Saturday
                  od
    end
```

This model is, in a sense, complete, but evidently it is not very useful: it is not a real-time model. It executes much faster than the real-world calendar, rather after the fashion of a spring-driven clock from which a naughty child has removed the escapement: as soon as we set it in motion, the wheels revolve without constraint, and hours pass in seconds. To make our model into a real-time model, we must slow it down. We therefore provide it with an input data stream, SRF, from which it will read a record each day. A record is written to SRF from outside the system at sunrise on each day. We may represent the configuration of the system diagrammatically as:



in which we follow the convention that circles represent data streams and rectangles represent processes.

The program text for WEEKCAL2 is as follows:

```
WEEKCAL2: begin
            open SRF; read SRF;
            while not eof(SRF)
                 do day:=Sunday; read SRF;
                    day:=Monday; read SRF;
                    ...
                    day:=Saturday; read SRF
                 od
            close SRF
          end
```

The records of SRF have no content; their purpose is purely to synchronise the model with the real world. Synchronisation is achieved because (a) the process WEEKCAL2 is blocked whenever it encounters a 'read SRF' operation and the next SRF record is not yet available (ie the sun has not yet risen on the next day), and (b) we assume that execution of a process which is not blocked is very fast in the context of the grain of time (in this case, a day) for which synchronisation is required.

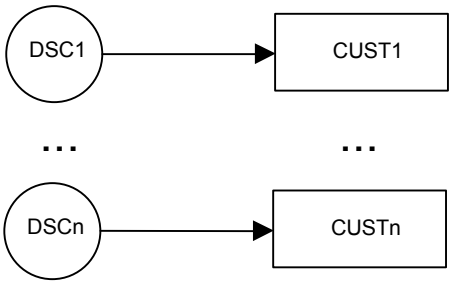
We may note in passing that a useful embedded function can be superimposed on this model by elaborating the process to produce a reminder each morning of the day of the week:

```
...
day:=Monday; print day; read SRF;
day:=Tuesday; print day; read SRF;
...
day:=Friday; print day; read SRF;
...
```

Example 3

A bank has a set of customers, each of whom has a balance in the bank's books. Each customer may incur debits (for example, by drawing cheques) and may effect credits (for example, by paying in cash or cheques).

The system consists of a process for each customer, each with its own data stream of input from the real world:



The grammar for the data streams DSC1, ... DSCn is:

```
<dsc>::= {<transaction>}*
<transaction>::= <credit>|<debit>
<credit>::= CR<amount>
<debit>::= DR<amount>
```

The program text for each customer is:

```
CUST: begin balance :=0;
            open DSC; read DSC;
            while not eof(DSC)
              do if CR
                    then balance:=balance+amount
                    else balance:=balance-amount
                 fi
              read DSC
            od close DSC
      end
```

Although there are several processes in the system, they are not connected in any way. Further, the timing of the individual processes is indeterminate, in that there is no well-defined grain of time to which the transactions can be allocated. For a superimposed function to report on the state of a single customer process, it will be convenient to make it an embedded function, reporting perhaps after each transaction or after any transaction which leaves the balance with a negative value. To report on the state of the whole system, however, an imposed function will be necessary: this imposed function, if it is to give valid results, must be executed when every process in the system is simultaneously blocked; otherwise the reported state of the system may be one which has never in fact occurred.

Example 4

The problem is the same as in Example 3, with the additional requirement that if, at the end of any day's business, a customer has a debit balance of £50 or more then an interest charge of one fiftieth of one per cent is added to the balance.

At first sight, it appears that we need only add a calendar input data stream SRFC to each customer process: SRFC contains one record for each day, available in real time; these records are marked with a day number, and the records of DSC are similarly marked, thus permitting the two data streams to be collated by the process.

However, there is an immediate difficulty. In order to collate the two data streams, the process CUST must be able to examine the next record on each; but when the next record of SRFC is for day(n), the next record on DSC may be for day(n) or for any later day. The process CUST is thus unable to apply the interest charge until some indefinite time has passed and a subsequent transaction has occurred.

This only matters in a real-time system. If we are willing to wait, possibly until the end-file marker appears on DSC (that is, until CUST1 has ceased to be a customer), we may be satisfied with the CUST process in the form of a simple collate with look-ahead on both input data streams. But this scheme does not satisfy even the weakest constraint of a real-time system: there is no point in time, t(i), earlier than termination of the whole system, at which we can guarantee that the state of the system will be up to date as at time t(j) for some value of j <= i.

Nor do we obtain relief from the general form of this difficulty by permitting a process to test the availability of a next record on an input data stream. Suppose, for instance, that we establish the rule

that any record of DSC for day(i) will be available before the SRFC record for day(i+l); the program text for CUST could then include something like:

```
    read SRFC; (*SRFC record for day(i+l)*)
if DSC record already read (*must be for day(i)*)
       then deal with DSC record
    fi
 X: while DSC record available
     _do read DSC;
         quit X if not record for day(i);
         deal with DSC record
    od (*end of X*)
    ...
```

Ignoring the unattractiveness of this rather disagreeable and messy text, we observe that the expedient cannot be extended readily to the general case. Suppose, for example, that the process CUST1 writes an output data stream which is read by another process P; and suppose that P, like CUST1, is required to carry out certain operations at the end of each business day, and, like CUST1, has an input data stream SRFP. We ought to be able to treat the process P in exactly the same way as we treated CUST1, but we cannot. CUST1 cannot write its output for day(i), in general, until it has seen all of its input for day(i); it can be sure of having seen all its day(i) input only after reading the SRFC record for day(i+l). P's input for day(i) is not therefore available before its SRFP record for day(i+l). We could repair the situation by staggering the different SRF data streams, so that the SRFP record for day(i+l) is available to P later than the SRFC record is available to CUST1, by an amount of time dependent on the speed of the CUST1 processor. Such a solution is clearly unattractive: its complexity will increase with the complexity of the system network, and it forces us to take account of the absolute and relative speeds of the processors.

We will not therefore permit any process to test availability of input records (a conclusion reached also by Kahn and MacQueen (4)). Instead, we will introduce marker records into the data streams to indicate the end of each time interval. Thus, in the present example, we will write 'end-of-day' marker records on the data stream DSC1; the grammar of DSC1 is now:

```
<dsc>::={<daygroup>}*
<daygroup>::=<daygroupbody> ENDDAY
<daygroupbody>::={<transaction>}*
<transaction>::= ...
```

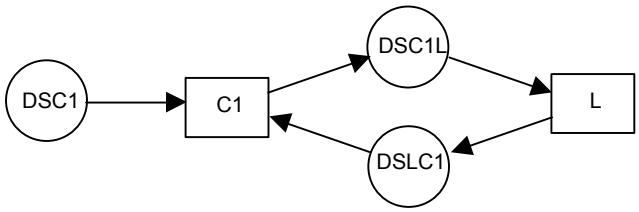
This data stream is readily collated with SRFC. Strictly, SRFC is now no longer required for correct execution of the process CUST1; however, it will often be advantageous to retain a calendar file such as SRFC to provide indications of time structure (such as months and years, or accounting periods) which are not conveniently indicated in other data streams.

Example 5

A lending library communicates with its customers by post. A customer may request a book; if it is available it will be sent to him, otherwise his name will be placed on a waiting list. After a customer has received, and subsequently returned, a requested book, he may request another book.

The system consists of a process for each customer and a process for each book; in addition, there will be a process for the librarian, whose responsibility it is to resolve conflicts among customers who request books on the same day.

The process for a single customer, say C1, has an input data stream DSC1, whose records are the customer actions RQ (request a book) and RN (return a book), interspersed with ED (end-of-day) records. It writes an output data stream DSClL which is input to the librarian process L: DSC1L is essentially a copy of DSC1, since the customer deals only through the librarian. The process L writes a data stream DSLC1 which is read by the process Cl: the records of DSLC1 are the librarian actions BS (book sent) and WL (wait-listed), and ED (end-of-day). A diagrammatic representation of the communication between the customer Cl and the librarian shows:



The grammar of DSC1 is:

```
<dsc>::=<inactiveperiod><dscbody>
<dscbody>::={<loan>}*
<loan>::= RQ <interval> RN <inactiveperiod>
<interval>::={ED}*
```

The process CI collates its input data streams DSC1 and DSLC1; as a result of this collation, a slightly more elaborate structure is imposed on <interval>, which is then divided into a <waitperiod> and a following <loanperiod>.

The restriction that a customer may not have more than one book on loan, or even request, at any time, makes it possible to design the system without a separate process for each loan. In a sense, there is a separate process for each loan, but it is a marsupial process: it remains in its mother's pouch. If the restriction is lifted, then the loan process must come outside the C process and live an !independent life of its own.

The librarian process has an input data stream for each customer in the system, and an output data stream to each customer in the system. Similarly, it has an input and output data stream for each book in the system. The program text of the librarian process expresses whatever algorithm is chosen for allocating books when there is conflict among customers. The process for each book has an input data stream from the librarian and an output data stream to the librarian. It also has its wait list, which can take the form of a data stream which is both output from and input to the book process.

Various embedded functions can be imposed on this system. The customer program may be equipped with embedded operations to allow it to acknowledge requests, inform the customer when a book is not available and the request is wait-listed, send reminders for overdue books, enquire of customers who are inactive for long periods whether they wish to retain their membership of the library, prepare mailing labels for mailing out books, acknowledge return of books, and so on. The book program may be equipped to estimate waiting time (from the number of records in its wait-list), keep statistics on the book's popularity and usage, remind the librarian when a certain number of loans have been made and the book is a candidate for rebinding, and so on.

4. TRANSFORMATIONS

4.1 Reducing the Number of Processors

Information systems do not conventionally have the appearance of the models described above. The most obvious difference is in the number of processors: instead of the hundreds, thousands or even millions of processors of our model, a conventional system has a single processor.

Transformation of a model system to execute on a single processor is straightforward, given a processor of adequate speed and storage capacity. The essential step is to provide a general-purpose operating system, which will manage the data stream queues and schedule execution of the individual processes by the single processor. To a substantial extent, such an operating system is provided in the execution-time support of those programming languages which permit the use of quasi-parallel processes.

Each individual process has an activation record and a re-entrant executable program text, the latter being shared with other processes of the same class. When a process is scheduled for execution, it is allowed to run until it encounters a read, write, open or close operation on a data stream, whereupon it is suspended. On suspension, its activation record will contain the current values of local variables,

including the text pointer; it is also convenient to record explicitly the type of operation and the identity of the data stream which have caused suspension.

Assuming a processor of adequate speed and storage capacity, any scheduling algorithm in the operating ,will suffice which does not permit reading from an empty data stream queue and does not for ever ignore a process waiting to read from a non-empty queue. For example, we might use the trivial algorithm:

```
while true
  do i:=1;
    while i <= maxprocesses
      do if process(i) not blocked
            then activate process(i)
         fi
         i:=i+1
      od
  od
```

In a real-time system in which the smallest grain of time is a day, to assume a processor of sufficient speed is to assume that each day the system would reach a *rest state* in which every process is blocked and further progress must wait until the next day's records become available in the external input data streams. The values of the activation records, in these rest states, depend only on the model system and its previous external inputs, and not on the scheduling algorithm or any other property of the operating system.

In conventional terminology, the activation record of, for instance, the customer process, is referred to as the 'customer record'. The value of the customer record at any rest state may be taken to represent the state of the customer: for example, the customer has submitted a request for a book, has been wait-listed, and has been waiting three days. Obviously, interpretation of the customer activation record depends on the customer program text, without which the customer activation record has no meaning.

If, as is usually the case, the storage required for the operating system, the model program texts and the process activation records exceeds the main storage available in the single machine, it becomes necessary to relegate model program texts and process activation records to backing storage such as disk, and to bring them into main storage only when they are needed. Hence an operating system must also, usually, include a program loader and a data management system.

4.2 Distributing the operating system

A general-purpose operating system of the kind described above is not likely to be satisfactory in practice. The overhead costs of process activation and suspension and of program loading and data management will be severe; also the execution times of the individual processes become important when there are very large numbers of processes sharing a single processor.

One task of the system designer may be regarded as the resolution of this difficulty by distributing :the operating system functions among the processes of the model system. In particular, the scheduling of processes can be wholly or partly determined at design time, and the scheduling decisions reflected in suitable transformations of the program texts. Consider, for example, the trivial case of a system consisting of only two processes:

We may decide that the process Q is to be activated whenever process P is suspended at an operation on the data stream DSQ. Q is then implemented as a semi-coroutine (6) in which the 'read DSQ' operations are compiled as 'detach' operations; P is implemented with its 'write record to DSQ' operations compiled as 'call Q' or 'resume Q' operations. Equivalently, Q may be compiled as an 'inverted program' (7) which appears to P as a 'dispose of record of DSQ' procedure; P invokes Q whenever P wishes to execute an operation on the data stream DSQ.

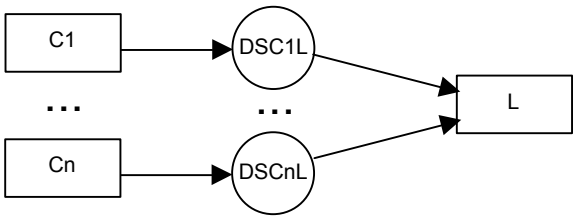
The effect of this transformation is that the data stream DSQ itself requires no identifiable implementation, and that the processes P and Q appear to the operating system as a single process. It is, of course, possible, and it may be attractive, to treat P itself in the same way as we are treating Q, whereupon the model system would appear to the operating system as a 'dispose of record of DSP' procedure: the operating system could then schedule the use of the processor by systems other than the combined pair of processes P and Q.

If, regarding each process and each data stream as a node, we can represent a system as a tree whose root is a process node, then the whole system can be reduced, by successive applications of the transformation described above, to a single process with a hierarchy of directly and indirectly invoked procedures.

4.3 Eliminating time interval markers

A glaring and intolerable inefficiency in our modelling of a real-time system such as that of the lending library was the introduction of the end-of-day marker records. Although it simplified the programming of the processes, and avoided the need to introduce any dependency on processor speeds, it also burdened the system with a plethora of additional records on each data stream: in many systems, these additional marker records would greatly outnumber the other records.

The markers can be eliminated by suitable scheduling of the processes; one method of scheduling leads to a batch processing system, and another method leads to what would be called a real-time system in a conventional sense of that term. With some simplification, both may be regarded as transformations of a part of the system which has the configuration:



The first transformation, leading to a batch processing system, is to ensure that the process L is not activated until the end of each time period, at which point the data streams DSClL, ... DSCnL will be known to contain all records generated by the processes C1, ... Cn for that period. The data streams may then be merged into whatever order corresponds to the order of the read operations in L. L can then read from this merged data stream with the normal look-ahead (of at least one record), and the absence of a potentially present record is detectable without any explicit end-markers other than the single end-file marker on the whole merged data stream.

The second transformation, leading to a real-time system in the conventional sense, requires that the process L be capable of accepting the records of its input data streams in whatever order they may be written by the processes Cl, ... Cn. L is implemented as an inverted program, appearing to the processes C1, ... Cn as a 'dispose of next record of DSCL' procedure; the merging of the individual data streams to give the single data stream DSCL is accomplished by permitting L to be invoked by all the processes which contribute to DSCL.

If we adhere strictly to the hypothesis that the grain of time is a day, then even in the transformed system L cannot allocate books until the end of the day. It is usual to abandon the grain of time altogether, permitting L to respond to a request record in DSCL before reading the next record of DSCL. The beneficial effect of fast response is then accompanied by two other effects: the possible algorithms for allocating books are severely curtailed, and the action of the system will in some cases depend on the comparative speeds of its constituent processes.

4.4 Program dismembering

Even when the overhead of process activation and suspension has been reduced by distributing the operating system among the processes of the model, there is still a motive for further transformation of the program texts.

When a process is activated to deal with a record of an input data stream, it will execute operations which depend, in general, on the values of variables including the text pointer and the data record. Either or both of these may be already known to the process writing the record, and some improvement in efficiency may be obtained by making direct use of this knowledge.

Consider, for example, the system of two processes P and Q discussed in section 4.2 above. Suppose that the grammar of DSQ, as seen by both processes, is:

```
<dsq>::= {<record>}*
<record>::= A|B
```

The program text of Q will be essentially the following:

```
Q: begin ..._read DSPQ;
     while not eof(DSPQ)
       do if A operation_a
          else operation_b
          fi
          read DSPQ
       od
   end
```

in which the type of record is evaluated to determine what operation should be carried out. But the program text of P will doubtless contain:

```
...
recordtype:=A; write record to DSPQ;
...
recordtype:=A; write record to DSPQ;
...
recordtype:=B; write record to DSPQ;
...
```

or something similar. There is an evident inefficiency in allowing process P to assign a value to recordtype merely so that Q can immediately examine that value. Instead, therefore, of keeping the text of Q as a single ‘dispose of next record of DSPQ’ procedure, we may dismember Q into its constituent parts and distribute these into the text of P. Thus we would replace the operations

```
recordtype:=A; write record to DSPQ;
```

in the text of P by the operation ‘operation_a’. and so on. The process Q would then effectively cease to exist as an identifiable process. It will also, of course, be necessary to replace the operation ‘open DSPQ’ in P by any operations which precede the loop in Q and the operation 'close ‘DSPQ’ by any operations which follow the loop in Q.

This dismembering transformation will not usually be so simple. Suppose, for example, that the grammar of DSPQ as seen by P is the trivial grammar above, but that the grammar as seen by Q is:

```
<qdspq>::= A{<record>}*
<record>::= A|B
```

and that the program text of Q is:

```
Q: begin ..._read DSPQ;
     operation_a1; read DSPQ;
     while not eof(DSPQ)
       do if A operation_a2
       ...
```

In the program text of Q, the determination whether to execute operation_a1 or operation_a2 is made automatically according to the value of the text pointer on activation. Once the program text of Q has been dismembered, P must make an explicit determination:

```
P: begin ...
     boolean firstA;
```

```
firstA:= true;
if firstA
   then begin operation_a1;
              firstA:=false
        end
   else operation_a2
fi
...
```

in which the boolean variable firstA is a vestigial remnant of the text pointer of Q, pointing to a program text which no longer exists.

An additional motive for program dismembering is due to the limitation of main storage available for program text. It is not usually possible to keep all program texts in main storage simultaneously, even after some have been dismembered. Since program loading is expensive in time, it is necessary to partition the program texts of the whole system into *load modules* which correspond approximately to the text to be executed on a particular activation of the system. In an on-line system, for example, the classical partitioning is into *transaction handling modules*. To achieve adequate response, the system is scheduled so that on receipt of a transaction record from an external data stream the reading process is immediately activated; as that process writes records to its output data streams the processes which read those data streams are activated; in turn they also produce records for which the reading processes are activated, and so on until the system is blocked. The program texts of the processes activated in this way are dismembered, and only those parts which can be executed directly or indirectly as a result of the incoming transaction type are included in the load module. The parts of the dismembered processes are thus distributed to many places in the resulting system.

4.5 Activation record handling

The retrieval of activation records and their retention is a function of the operating system which is readily distributed among the processes of the model: each program text is made responsible for identifying the particular process which is being activated and for retrieving and updating the activation record explicitly.

Just as program texts may be dismembered to allow efficient program loading, so too activation records may be dismembered to allow efficient data management. Instead of keeping the activation record of a process as a single object within the data management scheme, the designer may fragment it into parts which can be more quickly retrieved from backing storage. These parts can be linked together in ways which reflect the scheduling of processes: broadly, where there is a data stream in the model system connecting two processes P and Q there will be a link between the fragments of the activation record of P and the fragments of the activation record of Q.

5. CONCLUSION

This paper has been a plea for the separation of two pairs of concerns: the separation of model from function, and the separation of design from implementation.

There is nothing new here. Separation of model from function is an almost necessary concomitant of simulation programming in one sphere and of database design in another; separation of design from implementation is advocated by many, especially in the fields of abstract data types (9) and program transformation (10).

The greatest difficulty appears to lie in the separation of design from implementation: the transformations involved are too arduous to be carried out by hand with reasonable effort and reasonable reliability, but the choice of transformation is too difficult to automate. Some interaction between the programmer and the machine seems desirable: the programmer's human intelligence chooses the transformations to apply, while the machine applies them with a reliable guarantee that correctness is preserved.

ACKNOWLEDGEMENTS

In a paper of this nature it is difficult to make proper acknowledgements: readers will be able to identify many sources from which ideas have been drawn consciously or unconsciously. The members of the IFIPS Working Group 2.3 on Programming Methodology have been a source of both enlightenment and stimulus.

REFERENCES

1 IEEE Transactions on Software Engineering; Special Collection on Requirement Analysis; SE-3, 1 January 1977.

2 Dahl O-J and Hoare C A R; Hierarchical Program Structures; in Structured Programming (O-J Dahl, E W Dijkstra and C A R Hoare) p219; Academic Press, 1972.

3 Dwyer B; On the Relationship between Serial and Random Processing; to be published, 1978.

4 Kahn G and MacQueen D; Coroutines and Networks of Parallel Processes; IRIA Rapport de Recherche No 202, November 1976.

5 C A R Hoare; Communicating Sequential Processes; CACM 1978.

6 Dahl O-J and Hoare C A R; op cit (2); pp175-220. 7 Jackson M A; Principles of Program Design; Academic Press, 1975.

8 Martin J; Computer Data-base Organisation; Prentice-Hall, 1977.

9 Guttag J V et al; The Design of Data Type Specifications; in Proc 2nd International Conference on Software Engineering pp4l4-420; IEEE, 1976.

10 Burstall R M and Darlington J; Some Transformations for Developing Recursive Programs; in Proc International Conference on Reliable Software pp465-472; Sigplan Notices 10,6, June 1975.