STRUCTURE-ORIENTED PROGRAMMING

Michael Jackson
Michael Jackson Systems Limited
17 Conduit Street
London W1R 9TD
England

1 Introductory Remarks on Structure and Method

The two programs 'skip' and 'abort' (in Dijkstra's notation) have no structure: they are atomic, and therefore have no parts to be brought into a relationship by a program structure. All other programs have structure. An 'unstructured program', consisting of assignment, goto, and input/output statements, some labelled, has at least the structure of a directed graph. A 'structured program' in the conventional sense has the structure of a hierarchy or, perhaps, of a tree. The question then is not whether our programs are to have structure: they will have structure whether we like it or not. The question is rather: what structure should a program have? And, of course, how should we compose our program structures?

In the conventional sense 'structured programming' is the composition of programs by stepwise refinement or top-down design. The structure of the method corresponds exactly to the structure of the programs it is used to build. Initially a root node is defined; at each subsequent step, a terminal node of the tree or hierarchy so far constructed is visited; if it is not to be a terminal node of the finished program, it is refined or decomposed as a construct (sequence, iteration, or selection) whose parts are its child nodes in the program structure. The traversal rule for stepwise or top-down design is that a parent must be visited before any of its children. For the fundamentalist version of the opposite method - bottom-up design - the traversal rule is that each child must be visited before its parent. In practice, some bottom-up design is used in a preliminary phase to establish a more reasonable and useful set of primitives towards which top-down design can then proceed.

But even this mitigation of the rigours of top-down and stepwise methods does not make them satisfactory. They are deeply unsatisfactory for many reasons. First, they assume that the important structure of the program is that if a hierarchy or tree, when other structures (for example, networks) may be more suitable either in general or in particular cases. Second, they force the programmer to make the hardest decisions first. The initial decision, how to decompose or refine the root node, is the hardest of all: it concerns the structure of the whole program, and it affects critically all that follows. If the refinement of the root node is wrong, the work that follows is largely wasted. Third, the method in itself offers no help or guidance to the programmer in arriving at a sound program structure. The programmer must guide himself, usually by intuition or by recognising similarities between the current problem and other problems previously solved. Fourth, the method militates against separation of design from implementation: the programmer is encouraged to see implementation in the chosen programming language as merely the final steps of refinement, elaborating nodes far from the root, but never recasting the higher levels of the structure. Fifth, at every step the programmer is considering the program itself - that is, considering the problem solution rather than the problem statement or the domain within which the problem statement has meaning.

Of course, these criticisms are aimed at a simplified - perhaps even a caricatured - version of stepwise refinement. Those who practise stepwise refinement never use it in its simplest form, but introduce other considerations to help them to make sound refinement decisions. And, increasingly, quite different methods come to replace stepwise refinement: notably, methods based on correctness proofs and methods based on transformations. I believe that the transformational approach holds great promise, chiefly because it offers a context in which the programmer can deal effectively with the several different - and possibly conflicting - structures that are relevant to the construction of a program. In a substantial program, such as a data processing system, it is necessary to consider at least these structures: the structure of the problem domain; the structure of the problem within its domain; the structure of an acceptable problem solution; and the structure of an efficiently executable program embodying the solution. An effective method must allow separate consideration of these structures so far as is possible, and an orderly progress from the first to the last. The key techniques are the composition of existing structures to give a new structure and the transformation from one structure into another.

An effective method must also minimise the need for iteration in program development. The earliest steps in development should concern those decisions that can be made most surely and easily: typically, these will be decisions about how to describe something that already exists in the problem domain. Each decision made should be explicit, with as few hidden implications as possible. Each decision should be subjected as early as possible to the test of a later step that may show it to be wrong. The more likely a decision is to be wrong, the less work should be required to undo it. I believe that these principles are embodied in the approach described in this paper.

2 Problem Class and Solution Class

The class of problem I wish to consider is that in which the problem domain exhibits ordering, either in time or in space, as an essential characteristic. For example, in compiler design the problem domain is the set of all possible program texts (including, of course, those containing syntactic and other errors). These program texts are linearly arranged, and this ordering is essential: trivially, the statement x:=y is essentially different from y:=x; more significantly, the language definition may require all identifiers to be declared before they are used. A very different example is a system for handling a company's sales. The domain of the problem is the real world outside the system, in which customers order, receive, and pay for goods: these events are ordered in time, and it is of essential significance whether a customer pays for goods before receiving them. Yet another example, where the ordering is more abstract, is the problem of calculating prime numbers. Here the domain may be taken to be the natural numbers, and the ordering is that induced by the successor function.

It is important to distinguish ordering in the problem domain from ordering in the finished, executable, program. A compiler in its executable form may consist of lexical analysis, syntax analysis, and code generation phases, executed in that order; evidently, this is not an ordering in the problem domain, but a result of some decisions about implementation issues. In a similar way, a program to calculate prime numbers may begin with some representation of the natural numbers in a certain range and then delete successively the multiples of each number not yet deleted (the Sieve of Eratosthenes). Here, the ordering by which the number 10 (a multiple of 2) is deleted before 9 (a multiple of 3), is a result of an implementation choice, not alone of an ordering

in the problem domain.

The solutions we shall seek are those in which problem domain orderings are represented by sequential processes. I hold that this is a very natural choice, and preferable to the alternatives for the problem class we are considering. Suppose, for example, that we are dealing with a problem domain consisting of students who enter for examinations, sit the examinations, and receive grades when their papers are marked. Then we might consider representing this external reality by three sets: those who have entered, those who have sat, and those who have been graded. We would need in addition to state rules governing the addition of members to sets as the system moves from one state to another - for example, that the set of those who have sat is always a superset of those who have entered. I am claiming that it is more natural to represent the events in the life of each student in a sequential process, showing in the most direct way that a student must have entered before he can sit, and must have sat before he can be graded. In a similar fashion, we would choose to represent the behaviour of a customer by a sequential process showing that the customer orders goods before receiving them.

Because the problem domain ordering is almost always a partial and not a total ordering, we will find that our solutions involve large numbers of processes: there will be one for each student, one for each customer, and so on. Inevitably, this presents an implementation problem. Few execution environments are capable of executing large numbers of concurrent sequential processes, and few or none are capable of executing processes whose lifetimes may be measured in years. One approach to this kind of problem is to set about constructing a suitable execution environment: in essence, this is what is done for those programming languages for which a substantial run-time environment is provided. Our approach here is different. We aim to construct programs that may be executed in whatever environment may be available, even if that environment offers only a single real processor implementing a single virtual processor. In the implementation stage of development we therefore expect to transform our solution from one in which there are large numbers of long-lived processes to one in which there are few processes - perhaps only one - implemented so that they can be suspended and resumed over a very long period. Some simple problems may require little or no solution transformation; others may require substantial transformation and even the construction of special-purpose scheduling processes; others still may lie between these extremes.

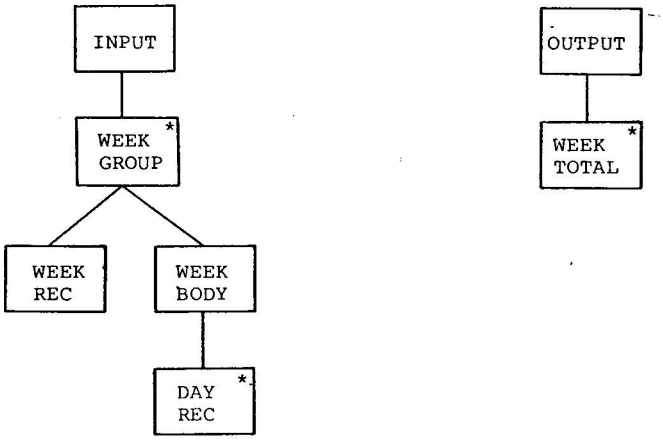
We choose this transformational approach to implementation because we do not wish to limit applicability of the method by the availability of particular execution environments. Nor do we want the execution environment to exert any strong influence on the early stages of development. It seems quite wrong, at the outset of a program development, to be forced to think in terms of a hierarchy of subroutines because the eventual implementation will be in Fortran; it is little better to be permitted to think in terms of coroutines because Simula is available as an implementation language.

3 Design of a Simple Sequential Process

Suppose that we are required to design and build a sequential process that has one input and one output stream. The input stream contains two types of record: WEEK records, containing the character W as a record discriminant and an integer WNO which is the ordinal week number counting from some defined starting date; and DAY records, containing the character D as discriminant and an integer DAMT which is the amount of

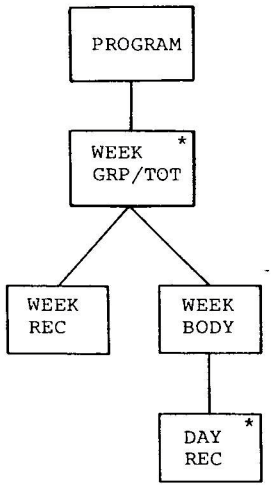
rain falling on that day, together with a second integer DNO which is the ordinal number of the day within the week (Monday = 1, and so on). The records are arranged in date order, the DAY records following their associated WEEK records. Where there has been no rainfall in a day or week the corresponding record is omitted. The output stream is to contain a record for each week in which rain has fallen, giving the total amount of rainfall in that week, the records being in date order.

We begin by describing the structures of the data streams in a diagrammatic notation:



The notation is largely self-explanatory. The asterisk (*) means iteration of zero or more occurrences; the parts of a sequence (WEEK GROUP in the example) appear in order from left to right. The structure we are imposing on the streams is roughly equivalent to a regular grammar, and some may prefer a textual notation to the diagrammatic notation used here.

We may regard the input and output structures as sequential processes, corresponding respectively to the ordered sets of events in the input and output subdomains of the problem domain. Our next step is to try to combine these two processes into a single process structure that will form the basis of the program. Evidently, the combined process structure should be:



The correctness of this structure, so far as it can be checked at this stage of the development, depends on its satisfying two criteria:

1 By suitable pruning the program structure can be reduced to either of the data structures.

2 Where a program component corresponds to a component in each data structure (as WEEK GRP/TOT corresponds to WEEK GROUP and to WEEK TOTAL), the number and order of occurrences of the data structure components is the same (there are the same number of WEEK GROUPs as of WEEK TOTALs, and they correspond pairwise).

The next step in development is to determine what executable operations are required for the program to produce its output from its input, and to embed those operations in the program structure. This step is very like embedding semantic routines in the structure of a parser. The essential criterion of correctness here is that the program structure should provide an obvious and natural location for each executable operation. For example, we can determine that a variable will be required to compute the WEEK TOTAL, and that this variable must be initialised by the operation tot:=0. The operation must be executed once for each WEEK TOTAL, at the beginning of the program component corresponding to that week; the appropriate place is clearly available, in the WEEK GRP/TOT component.

The fourth step is to consider whether there is difficulty in parsing the input stream, and in this case there is none (on the assumption that there will be single look-ahead). In more complex problems, it may be necessary to use multiple look-ahead or backtracking. In this brief account of the program development method we will not explore this aspect, except to say that the work that has been done on compiler constructions is clearly relevant.

Finally, we must convert the program structure, with embedded operations and specified conditions on the iteration and selection constructs, into a program text in a suitable language. In this tiny example, there is no difficulty in transcribing into any procedural language such as Pascal, Fortran, COBOL, PL/I, Assembler, or even Basic.

Reviewing what we have described so briefly, we may observe that:

- We began by describing the problem domain. Initially we did not even take account of the requirement that one of the streams was to be produced from the other.
- Because the problem domain was already defined, albeit informally, defining the stream structures was a task of description, not of invention.
- The program structure was formed by composition of the previously defined data structures, not by decomposition of the problem itself.
- The composition of the program structure could be considered as itself a transformation of the skeleton CSP program consisting of the two data structures as parallel processes.
- Some intermediate criteria of correctness are available, which will reduce the extent of the iteration necessary in design.
- Difficulty in parsing the input stream does not lead us to rewrite the grammar but rather to preserve the original structure and modify the mechanism used for recognition.

4 A More Complex Example

In the small example of the previous section we took the problem domain to be fully defined by the input and output data streams. We might have made more of the fact that there is a real world outside the system, abstractly described in terms of a simple calendar and the incidence of rain; and that the events of this real world - the arrival of a new day or week, and the falling of a unit of rain - have been captured in the program's input stream. But this would have been rather artificial: it was enough to take the data streams themselves as the problem domain.

We now look at a slightly more elaborate example, in which the problem domain is not fully described in any data stream, and the solution will demand more than one sequential process. The problem is taken from Dijkstra, who attributes it to Hamming: I have made a very slight change to the way in which the problem is stated.

A sequence of integers, S, is in increasing order. The members of S are defined by the axioms:

Axiom 1: 1 is a member of S.
Axiom 2: If n is a member of S, then so are 2*n, 3*n, and 5*n.
Axiom 3: S has no members other than by virtue of Axioms 1 and 2.

We have not stated the problem: only the problem domain, which is the sequence S. The problem may be simply to generate S, or it may be to determine how many members of S are less than 1000, or, generally, to answer any question that may be asked about S. Nonetheless, we can proceed on the basis that we must certainly construct at least a sequential process whose ordered events correspond to the members of S.

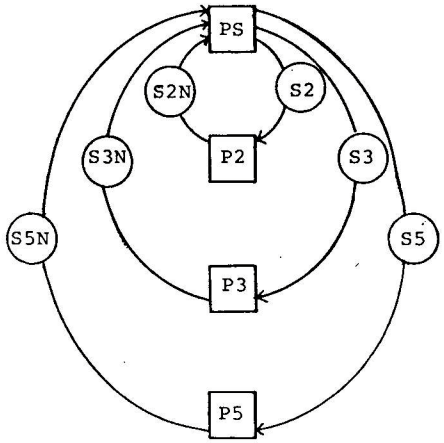
If we have only Axiom 1 and, mutatis mutandis, Axiom 3, this requirement would be satisfied by the process:

```
PS  seq
    l;
PS  end
```

Taking account of Axiom 2, we require three processes pi (i = 1,2,3) which take as input the successive members of S and produce as output those numbers multiplied by i:

```
Pi  itr while true
    read n;
    write i*n;
Pi  end
```

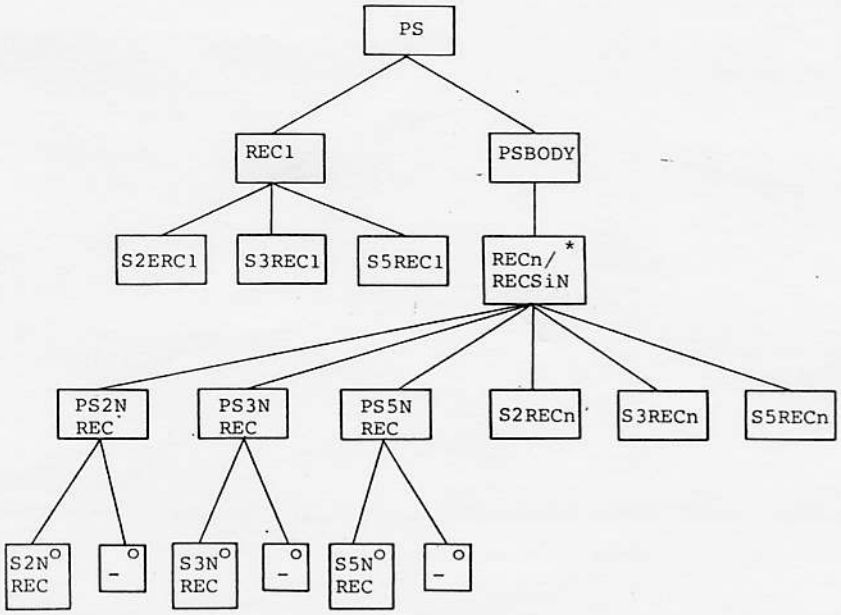
These processes must be connected to PS, which must be elaborated to write and read the communicating data streams, and to ensure the specified ordering of the members of S and the absence of duplicates. The resulting configuration is:



in which data streams are represented by circles and processes by rectangles. The streams S2, S3, S5 are identical copies of S. The rules governing the form of data stream (message passing) communication used here are:

- a stream has one writer and one reader process;
- a stream is unboundedly buffered;
- a stream is strictly sequential: the jth read operation by the reader process obtains the record written by the jth write operation executed by the writer process;
- write operations do not cause process blocking;
- the jth read operation on a stream causes blocking of the reader process if the writer process has not yet executed the jth writer operation.

A usable structure for the elaborated PS is shown below. It is somewhat optimised, but that is not our immediate concern here.



The components PSiNREC are selections, each consisting either of a SiNREC or of nothing (-). In text form:

```
PS   seq
     write 1 to S2; write 1 to S3; write 1 to S5;
     read S2N; read S3N; read S5N;
  PSBODY  itr while true
          n:=min(S2NREC, S3NREC, S5NREC);
     PS2NREC  sel (n=S2NREC)
              read S2N;
     PS2NREC  end
     PS3NREC  sel (n=S3NREC)
              read S3N;
     PS3NREC  end
     PS5NREC  sel (n=S5NREC)
              read S5N;
     PS5NREC  end
              write n to S2; write n to S3; write n to S5;
  PSBODY  end
PS   end
```

(We have, of course, omitted the substance of the design steps and, with it, the arguments that convince us of the correctness of our solution.)

The resulting configuration of four concurrent sequential processes may be executed in an environment providing implementation of the communication primitives and necessary four processors. However, we are more interested in seeing that suitable transformations can reduce our solution to a single sequential process capable of execution in a more general environment.

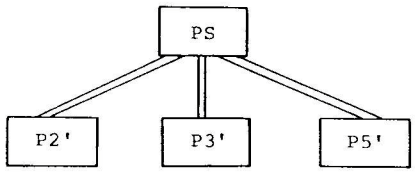
The essence of such a reduction lies in determining a scheduling of the four processes that will not preclude execution (for instance, by introducing deadlock or starvation) and can be bound somehow into the implemented program text. There are several possibilities, but here we consider only one.

In the unreduced solution, we may imagine each process Pi (i=1, 2, 3) running 'fast' or 'slow'. Running fast means producing its outputs on the stream SiN at the earliest possible moment by consuming each input record of Si as soon as it becomes available. Running slow means producing its outputs on the stream SiN as late as possible - that is, when PS is blocked at a read operation on SiN. We will choose to run the processes Pi as slow as possible. This choice has the advantage that the buffering requirement will be for the streams S2, S3, and S5 rather than for S2N, S3N, and S5N: since S2, S3, and S5 are identical, we may hope to achieve some useful economy of space.

Our scheduling choice can be implemented by converting each of the processes Pi into a procedure invoked by PS. This transformation consists simply of saving the state of Pi at each operation read Si, and returning to the invoking PS; the write Si operations in PS are implemented as invocations of Pi. This is the transformation that is sometimes known as 'program inversion'; it is broadly equivalent to casting each Pi in the form of a semi-coroutine. The effect is that execution of PS is suspended at each write Si operation; Pi is then activated, and suspended at its next read Si operation, whereupon execution of PS is resumed.

Having bound the mutual scheduling of PS and the Pi in this way, we can easily implement the streams Si as a single array, with suitable assignments to indices I1, I2, I3, and I5 as implementations of the write and read operations. The correctness of this implementation relies on our having already bound the process scheduling.

The transformations mentioned above have converted a network structure of four processes into a tree structure of a process PS and three procedures invoked by PS. This conversion from process network to procedure tree or hierarchy is typical. We may view the procedure mechanism - with own variables for the saved state - as being primarily a scheduling device: a process (in our example, each of the Pi) is transformed to a procedure so that another procedure or a process may control its scheduling. The resulting procedure hierarchy implements choices about control of scheduling, not about levels of abstraction of any other consideration belonging to the problem domain:



5 A Small Control Example

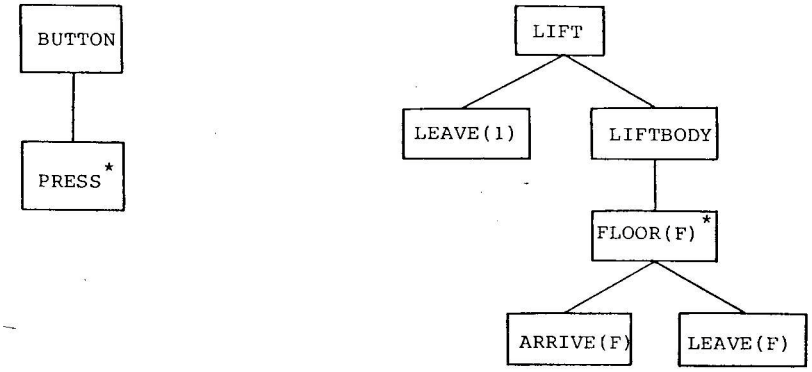
The problem domain for the problem of the preceding section was a mathematical construct, the sequence S. In this section we look at a problem whose domain is very much in the 'real world'.

The problem is to control an elevator in a building. The elevator is suspended from a motor-driven winch, the motor being capable of responding to the commands START, STOP, UP, and DOWN; START and STOP have their obvious meanings, and UP and DOWN set the direction of travel of the elevator for the next START command. At each floor there are buttons to summon the elevator for upwards and downwards travel (only the former at the ground floor, and only the latter at the top floor); inside the elevator there is a button for each floor to direct the elevator to travel to that floor. At each floor there is a sensor switch that assumes the 'closed' position when the elevator is within 10cm of its rest position at the floor, and is otherwise in its 'open' position.

We begin our development by considering what events in the problem domain we wish to describe and to model. We might, for example, wish to model the event 'passenger P enters the elevator', or the event elevator arrives at floor F'. Our purpose is to construct, as the basis for our system, a work-purpose is to construct, as the basis for our system, a working model of the problem domain that is rich enough to support the outputs that will be required. This working model will be driven by inputs from the problem domain to the system, these inputs being regarded as conveying information that something has happened in the 'real world' that must be replicated in the model if the model is to remain true to reality.

In this problem, we are heavily constrained by the available input mechanisms, assuming that we cannot add to those already described. We might, for example, wish to provide a very clever control system that cancels a command to travel to a floor if the passenger issuing the command leaves the elevator prematurely. But such a system would require inputs that refer to particular passengers; and that in turn would require the use of passenger badges and badge readers to supplement or replace the simple buttons already available. So we must limit ourselves to those events that can be detected and signalled by the input mechanisms we have. After some consideration we arrive at the very short list: ARRIVE(F); LEAVE(F); PRESS(B); meaning respectively that the elevator arrives at floor F, the elevator leaves floor F, and button B is pressed.

The time ordering of these events can be shown in our usual diagrammatic notation:



There is, of course, a constraint on the LIFT that is not shown in the diagram: the value of F in FLOOR(F) must always be G±1, where G is the value of F in the most recent LEAVE(F) event, and F must remain within the range given by the numbering of the floors of the building.

The tree structures BUTTON and LIFT represent both the behaviour of the real world buttons and elevator and the behaviour of the processes that will model them within the system. We will connect the model processes to the real world objects that they model in the obvious ways: the LIFT process will be decorated with suitable operations repeatedly examining the state of the floor sensors; the BUTTON processes, we will assume, receive input messages (electrical pulses) when the modelled buttons are pressed.

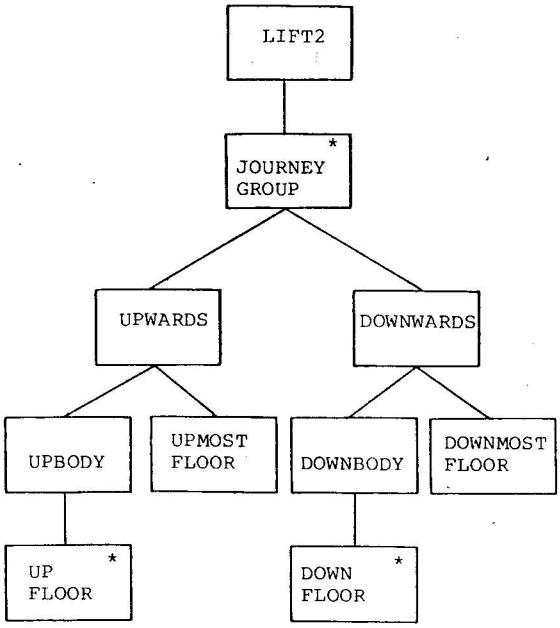
We have now specified our working model of the problem domain and can proceed to specify some of the desired functions of the system - that is, some of its desired outputs. Suppose, for example, that the elevator is equipped with a set of lights, one for each floor, intended to show the current position of the elevator. These lights can be turned on and off by ON(F) and OFF(F) commands, and are initially off. The desired outputs can be specified in terms of the events within the model process LIFT. Initially, the command ON(1) is output; when the elevator performs the action LEAVE(1), the command OFF(1) is output; when it performs the actions ARRIVE(F) and LEAVE(F) the commands ON(F) and OFF(F) respectively are output. Another example of a simple output requirement is an enquiry facility for anxious passengers waiting at the ground floor: by pressing a special button they can obtain a display showing which floor the elevator is at, or, if it is not at any floor, which floor it has most recently passed. To provide this output we add a process to the specification whose input is the stream of press operations on the special button; on receiving such an input message, the process inspects the state of the LIFT process and informs the anxious passenger of the current value of F.

The relationship between the model and the outputs is direct: outputs are obtained by operations embedded in the model or by inspection of the model state. In either case, the required output is specified in terms of the model itself. The adequacy of the model is measured by its

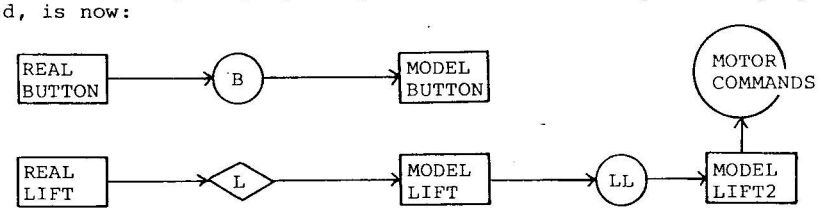
capacity to provide the basis for the functional specification, and the functional specifications are translated directly into elaborations of the system whose unelaborated form is the pure model.

The most interesting outputs, of course, are the motor control commands START, STOP, UP, and DOWN. For a clearer exposition of the model's relationship to the system functions we will specify these outputs in two stages. In the first stage we will specify that the elevator is to travel continually from the ground to the top floor and back again, irrespective of whether any buttons have been pressed. In the second stage we will take account of the buttons.

For the first stage, we require to show a structure for the elevator such as:

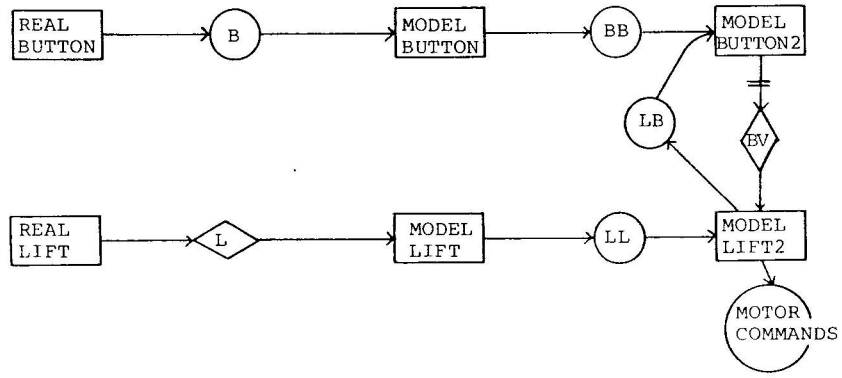


This is different from the previous structure given for LIFT, and we will embody it in a new process rather than elaborating the original structure. The new process will be connected to the original LIFT process by a data stream containing messages indicating that the elevator has performed a LEAVE(F) action, to allow the output of motor control commands to be properly synchronised with the behaviour of the real elevator. Our system, ignoring the small functions previously specified, is now:

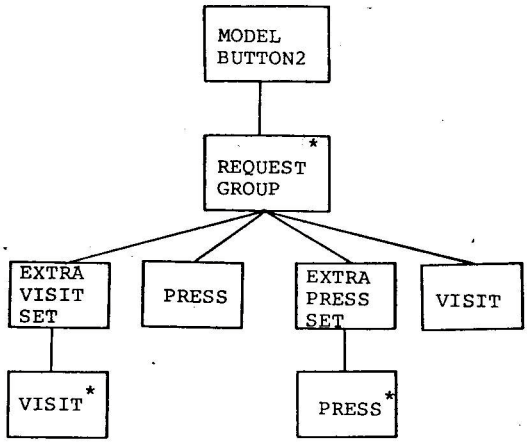


The REAL BUTTON and MODEL BUTTON processes are in 1-1 correspondence; there are as many of each as there are buttons (for example, 16 if the building has 6 floors). There is one REAL LIFT and one MODEL LIFT and one MODEL LIFT2 process. The connection between REAL LIFT and MODEL LIFT, shown in the diagram by the diamond marked L, is that MODEL LIFT inspects the state of REAL LIFT as given by the states of the floor sensors. All other process connections are by message streams, shown as before by circles. The MOTOR COMMANDS output is considered to be a message stream in which the UP, DOWN, START and STOP commands are transmitted sequentially to the motor.

For the second stage, in which the travel of the elevator is conditional on the pressing of buttons, we will need a more elaborate model of the buttons, to show whether a button has been PRESSed since the elevator last visited the associated floor. We will introduce MODEL BUTTON2 processes, whose input is a merging of the real world PRESS events sequence and of a stream of messages from MODEL LIFT2 indicating that the elevator has stopped at the floor. Additionally, we will specify that the MODEL LIFT2 process inspects the state of the MODEL BUTTON2 processes to determine whether or not to stop at a particular floor and whether or not to complete a full UPWARDS or DOWNWARDS part of a JOURNEY GROUP. The process connections are now:



The double bars on the MODEL BUTTON2 side of the connections between the MODEL LIFT2 and the MODEL BUTTON2 processes indicate that MODEL BUTTON2 and MODEL LIFT2 are in many-to-1 correspondence. Associated with this elaboration of the process connections are more elaborate process structures for MODEL BUTTON2 and MODEL LIFT2. The elaboration of the latter will depend on the algorithm chosen for controlling travel of the elevator: the choice of this algorithm is a matter for experts in traffic control, not for experts in computer system development, although its expression is certainly a task for the system developer. The structure of the MODEL BUTTON2 processes is inevitably:



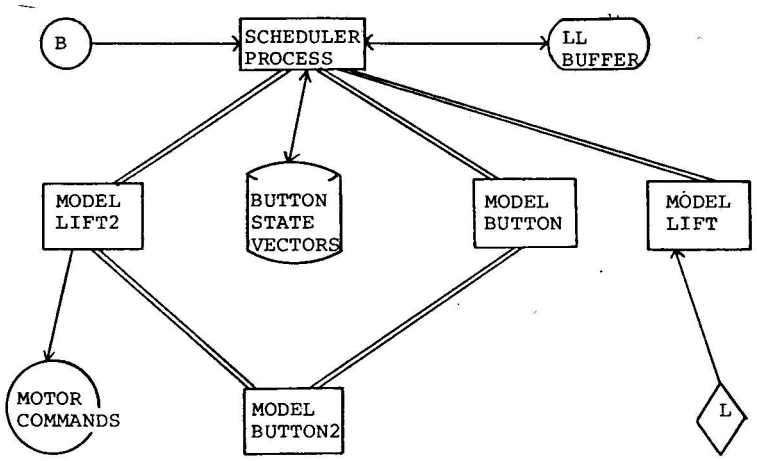
This structure accommodates all possible mergings of the BB and LB message streams for the button. We make no assumptions about synchronisation between the MODEL LIFT2 and MODEL BUTTON2 processes. The state of the MODEL BUTTON2 processes must include a variable OUTSTANDING-REQUEST whose value is initially false, set to true on the lone PRESS and reset to false on the lone VISIT event. It is this variable that MODEL LIFT2 inspects to determine the movements required of the elevator.

There is considerable indeterminacy in our specification. The sources of indeterminacy are the lack of specification of process scheduling, the connections by state inspection, and the merging of the BB and LB message streams. It is typical of both control and data processing systems that this indeterminacy is acceptable provided merely that it is bounded: we assume a reasonable scheduling of the processes, and a reasonable speed of transmission of messages in message streams. A full account of the development method would include a discussion of a step - the System Timing step - in which bounds on the acceptable indeterminacy are explicitly considered and formulated before the implementation of the system is designed.

A wide range of possible implementations is available. We concentrate attention on questions of process scheduling and the sharing of one or more processes of the specification. For each processor we design a procedure hierarchy, as in the preceding section. If for our present problem we assume an implementation on one processor we must design one procedure hierarchy to schedule and embody all the specification pro-

cesses. Relevant transformations in the present problem include both 'program inversion', briefly discussed in the preceding section, and the separation of a process' state from its program text to allow a single copy of the executable text to service many process instances. Separated state vectors become records in main or backing storage, arranged in structures such as arrays, trees, lists, queues or stacks designed to speed access to the process' state vector when the process is to be activated. Implementation of message streams may require explicit buffering mechanisms in some cases, although it is often an objective of implementation design to avoid buffering so far as possible.

An appropriate procedure hierarchy for a single-processor implementation is shown below. Symbols have been added to show the presence of the set of separated state vectors for the button processes and the buffering of (no more than one record of) the LL message stream. The inputs and outputs of the whole system have also been shown.



The SCHEDULER process is new, introduced only at the implementation stage. Whereas in the problem of the preceding section it was possible to use one of the specification processes (PS) as the scheduler, this is not possible in the present problem and a separate explicit scheduler is introduced. The structure of the SCHEDULER process expresses purely implementation decisions, concerned with the scheduling of the processes of the specification. In effect, it is the cyclic control algorithm with which development might - quite wrongly - have begun. The other processes in the diagram are transformations of the specification processes, these transformations being relatively simple and capable of mechanisation.

6 Data Processing and Other Systems

What distinguishes a system from a program? How is the task of system development different from that of program development? Paradoxically, I think that an enumeration of differences leads chiefly to a recognition of essential similarity (I do not quite dare to say 'identity').

First, the system development task usually includes the formulation of a specification, while the programming task usually begins from a given specification. Some would say that the programming task cannot begin until a perfectly exact and complete specification is available. This is an undeniable difference, although perhaps it is not very clear-cut. I would regard a specification as having two major parts:

1. A specification of the problem domain. For the class of problem I am considering, this specification will often best be expressed as a set of sequential processes, in some acceptable notation, describing the events of interest in the domain and their ordering.
2. A specification of the required program or system outputs. This specification has meaning only in the context of an agreed domain specification. It is typical of data processing systems that output specifications are changed and amplified over the whole life of the system.

Second, the problem domain for a program is usually either a mathematical abstraction or a predefined set of input and output data streams. The problem domain for a system is usually some part of the real world: the company's customers, suppliers, employees, goods, factories, warehouses; the airline's planes, flights, passengers, seats; the chemical plant's vessels, pipes, valves, fluids, gases; the motor car's engine, brakes, carburettor, gearbox. Such a real world domain must usually be described de novo. There is no mathematical library that may be consulted for an axiomatic description of the domain. It is also often permissible for the developer to ask for changes in the problem domain: the company's customers may be asked to act differently when the new sales order processing system is installed.

Third, systems usually involve long-running processes. The process describing the behaviour of a company employee must in general take many years to run to completion: useful data processing systems are always real-time systems in this sense. Implementation of systems containing long-running processes demands a special class of transformations. The state of each process must be held as a data object capable of retaining its value when the execution machine is switched off. Usually these process states are implemented as records in a set of 'master files' or in a 'database'. Activation and suspension of a process requires that the process state be retrieved and passed as a parameter to the executable process text, and stored in its updated form in the 'database' when the process is suspended.

Fifth, systems usually have so many processes communicating in a network of such complexity that it is necessary to construct special-purpose scheduling processes. From this point of view, the difference between a 'batch processing' system and an 'on-line' system is primarily a difference between their scheduling schemes. The task of devising scheduling processes is often mistakenly referred to as 'system design', when it would more properly be seen as an aspect of system implementation. Worse, it is then performed at an early stage of development, when the processes to be scheduled are not yet known. This error is commonly made in the development of many kinds of system, and may be attributed to a misunderstanding of the different roles of procedures and processes.

7 Some Concluding Remarks

The approach sketched out above is more fully described in books listed in the references. Some of the important points mentioned are these:

- Program development is based on processes rather than on procedures.
- The structure of a sequential process reflects the structures of its input and output streams.
- Process communication is by writing and reading sequential message streams.
- The specification of the problem domain is captured before the problem itself - the 'function' of the program - is considered explicitly or in detail.
- A central aspect of implementation is process scheduling. Often this will require transformation from a process network structure to a procedure hierarchy structure.
- Development of systems is a larger task than program development, but is now radically different.
- What is sometimes seen as the 'high-level' structure of a system is in fact a result of scheduling choices, and should emerge from the implementation, not from the early design stages.

The idea of defining a process within the system to correspond to an object in the real world - to an employee or a customer, for example - clearly has something in common with the ideas of data abstraction. The process state corresponds to the data values hidden by the abstract data type. A major difference is that abstract data type objects, as usually defined, fit into a procedure hierarchy rather than into a process network. We have regarded procedure hierarchies as a result of the implementation of process scheduling choices.

References

The items in the following list are not cited in the text of the paper, but are relevant to its themes.

(1) Some Transformations for Developing Recursive Programs; R M Burstall and J Darlington; Proc International Conference on Reliable Software 1975.

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

(3) A Discipline of Programming; E W Dijkstra; Prentice-Hall 1976.

(4) Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept; W M Gentleman; Software Practice and Experience, May 1981.

(5) The Design of Data Type Specifications; J V Guttag, E Horowitz, and J D Musser; in Current Trends in Programming Methodology IV; ed R T Yeh; Prentice-Hall 1978.

(6) Communicating Sequential Processes; C A R Hoare; Comm ACM August 1978.

(7) Principles of Program Design; M A Jackson; Academic Press 1975.

(8) System Development; M A Jackson; Prentice-Hall 1982.

(9) Coroutines and Networks of Parallel Processes; G Kahn and D McQueen; INRIA Research Report No 202 November 1976.