Tools and notions for program construction: An advanced course;
Nice 1981; pages 1-25; Cambridge University Press, 1982.

A SYSTEM DEVELOPMENT METHOD

M. A. Jackson
Michael Jackson Systems Limited
101 Hamilton Terrace, London NW8 9QX, England

1. Introduction

1.1 The nature of method

A development method may be regarded as a path or a procedure by which the developer proceeds from a problem of a certain class to a solution of a certain class. In trivial cases, the method may be fully algorithmic; for example, there is an algorithmic procedure for obtaining the square root of a non-negative number to any desired degree of accuracy. In more interesting cases, such as the development of computer-based systems for purposes such as data processing or process control, we do not expect to find an algorithmic method: the goal of the development is not precisely defined, and neither the problem nor the set of possible solutions is sufficiently well understood. But a method, to be worthy of the name, must at least decompose the development task into a number of reasonably well-defined steps which the developer can take with some confidence that they are leading to a satisfactory solution.

The steps-of a method impose some ordering on the decisions to be taken during development . Here we are using the term "decision" in a wide sense: it may denote a decision to state explicitly some *a priori* truth about the subject matter of the system, or a decision to define a certain abstract data type, or a decision to store certain variables on disk, or a decision to decompose some system function into three parts, or any of a wide range of possible actions whose result contributes to the final solution. Some decisions will be easy to take, especially where they involve little more than the recording of already known facts: for example, the decision that a multiple of a prime number cannot itself be a prime number. Some will be hard to take, perhaps because they require great foresight on the part of the developer. Some will be highly error prone, and some will be taken in confidence of their correctness. Some will have very limited consequences for the work that follows, some will have wide consequences.

From this viewpoint, we can discern some inchoate principles of methodology (a much misused word, denoting the study or science of method). Decisions about the subject matter of the computation should be taken before decisions about the computation itself. Decisions about implementation should be taken after decisions about what is to be implemented. Decisions which are error-prone should be taken as late as possible, when the developer is most likely to make them correctly. Decisions which have wide consequences should not be highly error-prone. The more error-prone a decision, the more important that the method should provide an early confirmation or disproof of its correctness.

These principles suggest strongly that those methods which can broadly be described as "top-down functional decomposition" are to be avoided. In such a method, the developer begins by deciding the structure of the highest level of the system, viewed as a hierarchy. This decision has the widest possible consequences, because it conditions everything that follows. It is highly error-prone, because it concerns the system itself, which, by definition, is as yet unknown. It is placed at the very start of the development procedure, but it may be invalidated at the end. We may suspect that developers who purport to be using such methods for development. are, in fact, using them only for description; the development work has already been largely done, informally and invisibly, in the developer's head.

1.2 Scope of this method

The method described in this paper is concerned with those problems for which the subject matter is strongly and inherently sequential, and especially those where the sequentiality is an ordering in time. We will refer to the subject matter of the system as the "real world" for the system. The real world for a payroll system contains the employees, their work, their holidays, their periods of sickness, their promotions, their productivity. The real world for a process control system contains the plant to be controlled, its vessels, pipes, valves, the substances being processed, the flows of those substances, their

temperatures and densities. The real world for a telephone switching system contains the subscribers, their telephone apparatus, the calls they make, the trunk lines and relays. For all o£ these, the real world is strongly time-ordered. The telephone subscriber goes off hook before dialling the first digit; he dials the area code before dialling the exchange number; he converses before going on-hook again. The employee joins the company before he works; he goes on holiday before he returns form holiday; he retires before he receives a pension. An adequate description of the real world must do justice to this time-ordering.

For some problems, which are outside the scope of JSD, there is no time-ordering in the real world. A classic example is the problem of providing a system to answer questions about the results of a national census. The real world for this system is a snapshot of the state of the nation at one moment in time, the moment when the census is taken. There is no time-ordering in this real world: there are no events occurring in sequence, only a state of reality at one moment. Of course, when the system has been constructed to run on a sequential machine, there will be time-ordering in the operation of the system itself; but this is an artefact of the development , not an aspect of the real world. JSD is concerned only with systems for which the real world is sequential.

1.3 Principles of this method

The first principle of JSD is that development must begin by describing and modelling the real world, not by specifying, describing or structuring the function which the system is to perform. A system developed according to the JSD method embodies an explicit simulation of the real world; this simulation is specified before any direct attention is paid to the function of the system. Of course, the developer must have in mind a general idea of the system's function and purpose, to guide the modelling activity; but, just as in the construction of a simulation program, the model is built first and the parts of the system which produce output are added later.

The decisions to be taken in the modelling steps of JSD are, in general, likely to be easier than decisions about the system function. Often (though not always) the real world already exists and is well known to the intending users of the system. In modelling this real world, the developer captures the user's view of the world they inhabit; although the developer must provide the power of abstraction, he will be able to look to the users for authoritative guidance in many of the decisions he must make. These decisions will therefore be easier and less prone to error. Later, when the system functions are specified, decisions will again be easier: an agreed model of the real world provides a sound basis for clear and unambiguous discussion of functions. We may go further: the explicit statement of a model defines a universe of possible functions; every function to be performed by the system must be capable of specification in terms of the model.

The second principle of JSD is that an adequate model o£ a time-ordered real world must itself be time-ordered. In particular, a database model of reality is not adequate. An appropriate modelling medium is provided by communicating sequential processes, in which the ordering of events in the real world is directly represented in program texts for the processes. We may adapt a statement made by Dijkstra in a different context (Structured Programming; Dahl, Dijkstra & Hoare; p 22):

> "... we should restrict ourselves in all humility to the most systematic sequencing mechanisms, ensuring that 'progress through the computation' is mapped on 'progress through the text' in the most straightforward manner."

Our purpose is to map progress through the real world on progress through the system that models it. A model consisting of sequential processes allows this to be done in the desired straightforward manner.

The third principle is that the system should be implemented by transforming the specification into an efficient and convenient set of processes, adapted to running on the available hardware and software. A central concern in the implementation stage is process scheduling; a small number of available processors must be shared among a large number of specification processes, and it is highly desirable to determine and bind the scheduling of processes when the system is constructed, rather than waiting until it is run. The transformations are therefore chiefly concerned with process activation and suspension.

The advantages of transformation as an implementation method are obvious, and have been explained often (see, for example, Some Transformations for Developing Recursive Programs; Burstall & Darlington; Sigplan Notices 10, 6, pp 465-472). JSD specifications are highly explicit; they describe very directly the real world to be modelled and the outputs to be derived from that model. Being in the form

of communicating sequential processes, they are amenable to a powerful yet simple set of transformations that suffice to give several different implementations of the same specification.

2. The Development Medium

2.1 Sequential Processes

A sequential process is an execution instance of a program text. The text is a tree structure of sequence, selection, iteration and elementary components; both diagrammatic and textual representations are used for these tree structures, but the usual diagrammatic representation is incomplete (it does not, for example, show the conditions on selections and iterations). By definition, there is only one instance of the program text pointer for a sequential process, and hence there is no concurrency or parallelism within one process.

The control flow constructs of sequence, selection and iteration are augmented by backtracking versions of the selection and iteration constructs. A *quit* statement is provided which may be written within the body of an iteration or within the first part of a two-part selection; execution of the quit statement transfers control to the end of the iteration component or to the beginning of the second part of the selection. Use of this quit statement, which is equivalent to a limited form of GOTO, is considered highly preferable to the use of local variables for the same purpose; control flow should be explicit, and should not be hidden in Boolean variables. The appropriate design discipline for use of these backtracking constructs is discussed in Principles of Program Design [M.A. Jackson; Academic Press, 1975], where it forms a part of a general design method for sequential processes.

2.2 Data Stream Communication

One process may communicate with another process by operations on a named data stream. The operations on a data stream are open, close, write, and read. For any data stream there is a writer process and a reader process. The writer process executes the operations

```
open, write, ..., write, close
```

and the reader process executes the operations

```
open, read, ..., read, close.
```

The close operation executed by the writer process embodies an implicit write operation which writes a special end-marker record.

The data stream is considered to be an unbounded buffer in which writing and reading obey the FIFO rule. When the reader process reaches a read operation in its text, and the data stream from which a record is to be read is empty (the number of records previously written is equal to the number previously read), the process is blocked; its execution cannot continue until another record is written to the data stream. A writer processes never blocked, because the buffering in the stream is unbounded.

A process which is the reader of two or more data streams may determine the interleaving of read operations on those streams by the sequencing of the read operations in its text. Alternatively the inter leaving may be determined by the order in which records become available on the data streams. The reader process of a data stream may test the value of a Boolean predicate empty, which indicates whether or not the stream is empty and hence whether a read operation would cause the process to become blocked. This interleaving scheme is called a rough merge; its effect depends in general on the implementation of the system, and thus introduces indeterminacy into the specification . Where a rough merge occurs in a JDS specification, the developer has the option of writing an explicit merging process using the empty predicate, or of leaving the merge implicit.

2.3 State Vector Communication

The state vector of a process consists of its local variables together with its text pointer. The value of a process's state vector can be changed only by execution of the process itself: it is strictly an own variable from this point of view.

In one respect the state vector of a process in JSD is not strictly an own variable: it can be directly inspected by other processes. A process P may execute the operation

```
getsv (Q)
```

Having executed this operation P may then inspect and use the current values of Q's local variables.

The value of a state vector obtained by a getsv operation is always a value at an open, write, read, close, or getsv operation in the process whose state vector is obtained. This rule may be satisfied by the use in Q of a private copy of its state vector, assignments to the public copy being made only at the appropriate points in its execution; although Q's state vector is a shared variable for P and Q, only the lowest-level mutual exclusion is required.

State vector communication, like rough merge, introduces indeterminacy into the specification. The value of Q's state vector obtained by P will depend on the implementation of the system; only the implementation will determine the relative scheduling of P and Q.

3. JSD Development Procedure

3.1 Development Steps

There are six steps in the JSD development procedure:

- Entity/Action step,
- Entity Structures step,
- Initial Model step,
- Function step,
- System Timing step and
- Implementation step.

In the Entity/Action and Entity Structures steps, an abstract description of the real world is made; the world is described in terms of entities which perform and suffer actions. The entities and their actions are listed in the Entity/Action step, and the constraints on the time ordering of the actions (for example, the constraint that a library book must be lent before it is returned) are expressed in the Entity Structures step. This abstract description is realised in the Initial Model step. For each structure specified in the Entity Structures step, a sequential process is specified and a mechanism for connecting that process, directly or indirectly, to the real world entity which it models. The connections between the real world and the model, like the internal connections of the system being developed are by data stream or state vector.

The Initial Model step specifies a simulation of the real world; the Function step adds to this simulation the further executable operations and processes needed to produce the outputs of the system. Because of the indeterminacy in the system, due to the rough merges and state vector connections which it may use, it is necessary to consider explicitly whether the values of system outputs are sufficiently constrained. For example, the user of an accounting system may be satisfied if answers to account enquiries are based on information that may be as much as one day out of date, or he may not. These constraints are partly specified in the Function step, by choosing appropriate connections among processes; they may require further consideration in the System Timing step, where tighter synchronisation among processes can be specified.

The specification produced at the end of the System Timing step is, in principle, capable of direct execution. The necessary environment would contain a processor for each process, a device equivalent to an unbounded buffer for each data stream, and some input and output devices where the system is connected to the real world. Such an environment could, of course, be provided by suitable software running on a sufficiently powerful machine. Sometimes, such direct execution of the specification will be possible, and may even be a reasonable choice. More often, the mismatch between the specification and the available hardware/software machine will be too severe. A data processing system specification may contain hundreds of millions of processes: a social security system for a nation must have at least one process for each member of the nation. But no available hardware system contains hundreds of millions of processors, nor is any available operating system capable of scheduling so many processes with reasonable efficiency.

It is therefore necessary to reduce the number of processes from the large number contained in the specification to a much smaller number, so that a real or virtual processor can be provided for each process to be executed. This is the central concern of the Implementation step. By suitable transformations, processes are combined so that their number is reduced to the number of processors: execution of the combined processes is interleaved, so that they appear to their processor to be a single process.

3.2 Entities and Actions

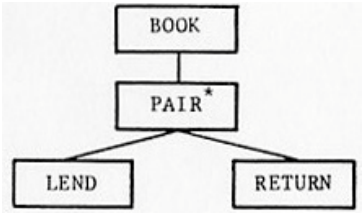
The ideas of entity and action are closely related in JSD. An action is an atomic event, occurring at some point in time, and considered to be instantaneous. One or more entities participate in an event, either performing or suffering the action. Over its lifetime each entity participates in a number of events, the ordering of the events being subject to certain predetermined constraints. This participation is the defining characteristic of an entity in JSD.

It is the real world which is described in terms of entities and actions, not the system itself. In the early steps of JSD development, attention is focused entirely on the real world outside the system. We are therefore careful to exclude from the lists of entities and actions any object or event which belongs to the system being developed rather than to its subject matter in the real world. Thus, for example, in developing a system to control a chemical plant we would regard the plant as a part of the real world, but the control mechanism as a part of the system; there would therefore be no entity "controller" in the abstract description of the real world, and there would be no action "output control signal to valve" or "turn on warning light".

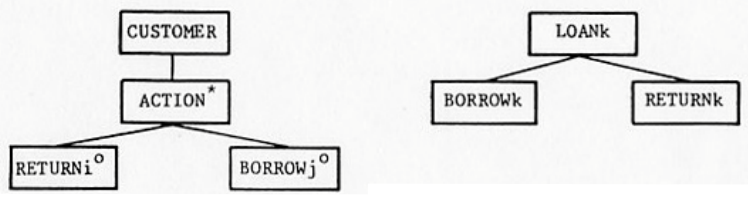
This view of entities and actions is close to the view underlying Hoare's CSP (Communicating. Sequential Processes; C.A.R. Hoare; CACM 21, 8, pp 666-677). It is radically different from the view taken in database design. Because a JSD entity must have a significant time-ordered history of actions, there will be many fewer entity types in a JSD description than in a typical database definition; a JSD entity does not correspond to an owner or member of a set in a network database, to a segment in a hierarchical database, or to a 3rd or 4th normal form relation in a relational database.

The guiding principle in choosing entities and actions is that the description must be able to support the functions to be provided by the system. Even when the system is intended to function as a control system, the criterion is simple: will the system need to produce or use information about this event or about the history of this person or object? Putting the same criterion in a negative form, if this putative action or entity is omitted, are there any required functions that cannot then be provided ?

In the Entity Structures step it may become apparent that more than one structure specification (and hence, eventually, more than one sequential process) is needed to describe the behaviour of an entity. The structure specifications are regular expressions, eventually to be translated directly into process structures. It will often happen that one regular expression is not enough to describe the constraints on the ordering of the actions of an entity. Consider, for example the following two cases. In the first case, we wish to describe the behaviour of a library book, whose actions are lend (i.e., the book is lent) and return (i.e., the book is returned). The necessary constraints are that lend and return actions must alternate, the first action of the book being a lend action. Diagrammatically we represent the lifetime of the book as:



The asterisk in the box PAIR indicates that BOOK is an iteration of (consists of zero or more instances of) PAIR. In the second case, we wish to describe the behaviour of a customer of the same library. The actions are BORROW and RETURN, with obvious meaning; the constraint is that each BORROW must precede the corresponding RETURN, but BORROW and RETURN need not alternate. We must represent this constraint by two structures:



The CUSTOMER structure shows that the behaviour of CUSTOMER is an iteration of ACTION, each ACTION being a selection of RETURN and BORROW (indicated by the circles in those boxes). Each

RETURN and BORROW is associated by its index with a particular LOAN and the behaviour of a LOAN is a sequence of BORROW followed by RETURN.

The developer might arrive at the two structures, CUSTOMER and LOAN, by either of two paths. He might start, as suggested above, with only CUSTOMER as an entity; in the Entity Structures step he is then forced to introduce the LOAN structure to describe the constraints on the ordering of CUSTOMER actions. Or he might start with both CUSTOMER and LOAN as entities, BORROW and RETURN being actions common to both.

3.3 Functions

The result of the Initial Model step is a specification of a workable simulation of the real world; for each structure showing the time-ordered actions of an entity, there is a sequential process driven by the real world actions. In the Function step, the developer adds to this simulation the operations and processes required to produce the system outputs.

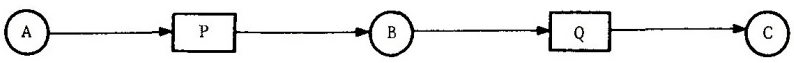
In the simplest case, an output may be provided simply by embedding a write operation in an appropriate process of the model. For example, an exception report to be produced whenever a customer has more than 10 books on loan can be embedded in the CUSTOMER process directly. A function specified in this way is called a simple embedded function.

A more complex function may require the addition of a new process to the specification, connected to one or more model processes by data stream or state vector connection. For example, a report listing the library's books and showing for each one how many times it has been lent would be provided in this way. Such a function may also require input, if the report is to be produced on demand. If the function process is connected by state vector connection to the model processes, it is called an imposed function; it uses the model by inspecting its state, and may be thought of as being imposed on the model as another level of the system.

The most complex functions are those which change the state of the model, as neither embedded nor imposed functions do. Suppose, for example, that a customer may perform the action of requesting a book which is out on loan, and that the system is to contain an allocation process which reserves the book for loan to that customer when it next becomes available. Then the state of the book process must be changed by the allocation process. The allocation process will he connected to the book process by two connections: it will inspect the state vector of the book process, and it will also write a data stream of reservation records which is input to the book process. The allocation process is called an interacting function because it interacts with the model instead of merely receiving input from it.

3.4 Process Scheduling

Suppose that in a very simple system we have a process P and a process Q, connected by a data stream B; P has an input data stream A and Q has an output data stream C. We represent this system diagrammatically as:



Suppose also that we wish to implement this system on a single processor, and must therefore combine P and Q into one process from the execution point of view. (We would, of course, regard the use of multi-programming or multi-tasking facilities as the use of two processors.)

To combine P and Q into a single process we must choose a scheduling scheme for sharing the processor between them, and implement this scheduling scheme as an explicit interleaving of the execution of P and Q. One possible scheme is to suspend and activate each process once for each record of B, and to activate them alternately. The necessary transformation of P and Q is to transform them into procedures which may be invoked respectively to produce and consume one record of B. This transformation is called "inversion with respect to B"; it consists essentially of implementing the "write B" and "read B" operations as:

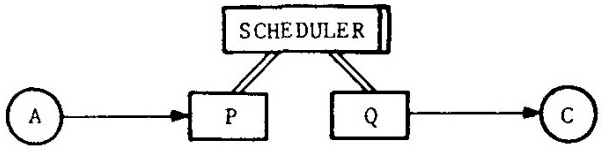
```
      save process text pointer;
      return to invoking program;
label:
```

where the saved value of the text pointer is the value of label. At the entry point of the transformed procedure is a suitable mechanism for resuming execution at the next invocation:

```
entry: GO TO (text pointer);
```

The process state is held in own variables within the process; the text pointer must be set to its initial value before process execution begins, and this can be done conveniently at compile time.

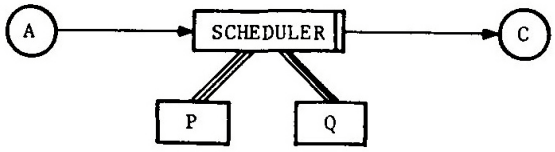
We now design a special-purpose scheduling process which invokes P and Q according to the chosen scheme:



The scheduler itself is, in outline

```
while true do
   call P;
   call Q;
od
```

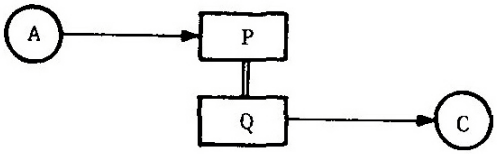
We could, of course, replace this scheduler by another which implemented a different scheme; for example, we might provide the scheduler with a buffer for 10 records of B, and activate P and Q alternately 10 times each. More ambitiously, we could invert P and Q with respect to all of their data streams:



The scheduler may now buffer records of A, B and C, and may therefore implement quite elaborate schemes. For example, it might buffer all records of A until a record of a certain type appears; then it might activate P and Q alternately, buffering the records of C; it might produce records of C in batches of 100. Batch data processing systems may be regarded as the product of this kind of implementation.

Notice that inverting a process with respect to more than one data stream requires a more complex interface with the scheduler: the scheduler must be able to determine whether the suspended process P is suspended at a “read A” or at a “write B” operation.

For the simplest scheme mentioned above, in which P and Q alternate on each record of B, we could have used one of the system processes itself as the scheduler:

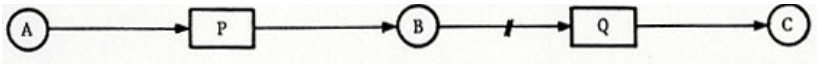


The “write B” operations in P are implemented as calls to Q. it is possible to implement systems of several processes in this way. A sufficient condition for such an implementation is: the system forms an acyclic undirected graph when each process is regarded as a node and each data stream connecting two processes is regarded as an arc.

3.5 State Vector Handling

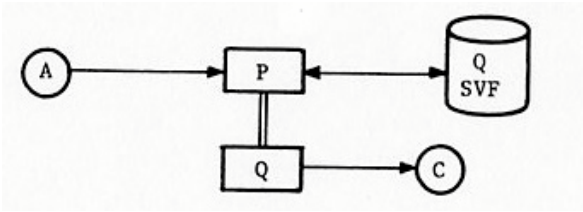
If there are several processes modelling several entities of one type, they will have a common program text. An obvious implementation device is to use only one copy of the program text and to associate the appropriate state vector with the text when a process is activated. In a data processing system, the state vectors of processes, treated in this way, become data records corresponding to the entities modelled by the processes. It is necessary to handle these state vectors explicitly as data objects in the scheduling processes.

Consider the very simple system discussed above, with the elaboration that there are multiple instances of the process Q and hence of the data stream B. Diagrammatically we represent this system as



The double bars on the arrow from B to Q indicate that the relationship between P and Q is one to many.

Suppose now that Q is inverted with respect to B, and that P is to be used as the scheduler process. The state vectors of the Q processes are held in an explicit direct-access store which is accessible to P:



Each "write Bi" operation in P is implemented as

```
obtain state vector of Qi;
call Q (Brec,state-vector);
replace state vector of Qi;
```

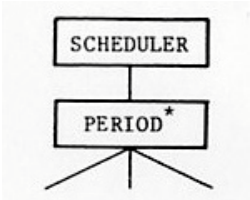
The file handling and data base accessing programs of a typical data processing system may be regarded in this light. The access paths to data records are then determined by the choice of process scheduling.

3.6 Process dismembering

All transformations considered so far have kept the program text of a process in one piece. Sometimes it is necessary to dismember the text of a process into several pieces for greater convenience of execution.

One important reason for process dismembering is the need to limit the elapsed execution time of the longest process. In a data processing system, the model processes will typically have execution times measured in years or even decades, because they model in real time the behaviour of real world entities whose lifetimes are long. The use of program inversion allows these processes to be suspended and reactivated almost at will, but this merely transfers the difficulty to the scheduling process. If the execution time of P and Q in the simple system discussed above is, say, ten years, we can break this execution time into small periods by inverting both P and Q and placing them under control of a scheduler. But now the scheduler has an execution time of ten years. This is clearly unacceptable in the execution environments available today.

The solution to the difficulty depends on the scheduler having a structure which admits of convenient decomposition. Suppose, for example that the scheduler has the structure:



in which one PERIOD component is executed, say, each day. Then we may dismember the scheduler into two parts: one part is the PERIOD component, and becomes a batch program or a day's running of an on-line system; the other part is the iteration SCHEDULER itself, which may be implemented by instructions to the computer operators ("each day, please run the program PERIOD"). Clearly, if the state vector of SCHEDULER contains variables which must be remembered from one PERIOD to another (i.e. are not local to the PERIOD component), provision must be made for the storage and retrieval of these variables between executions of PERIOD.

4. A Simple Example

4.1 The starting point

A system is required to control the operation of a simple elevator in a building. The building has six floors, and there is one elevator shaft only. The control requirement is to provide a reasonable service to people who want to use the elevator to travel from floor to floor. Some other functions may be needed, but nothing further is yet known.

This starting point, of course, is not a specification; nor should it be. Our purpose in JSD is to develop the specification, no less than to develop a system to satisfy it.

4.2 Entities and Actions

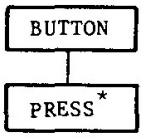
Examining the real world with which the system will be concerned, we may identify several candidate entities: the elevator, the shaft, the floors, the people who use the elevator, the motor which raises and lowers the elevator, the call buttons, the various lamps, the. sensors which sense whether the elevator is positioned at a floor. Applying the criterion discussed in section 3.2 above, we decide that we are interested only in the elevator and the people: we do not expect to provide functions which will produce or require information about the behaviour of the motor, the lamps, the buttons, or the sensors. We expect to provide functions which can be expressed in terms of the actions of the elevator and its users: "when the elevator reaches a floor, if some person wishes to enter or leave at that floor, then ...".

However, some intelligent anticipation suggests that we should not choose "person" as an entity type, because it will not be possible to identify individual people and their actions: to do so, we would need to provide users with identity cards, and require them to insert these cards in a slot beside the call button before pressing the button, since only in this way could we obtain the information necessary to model the behaviour of each user correctly. We therefore decide to abandon "person" as an entity type, and to reintroduce "button"; in some sense, each button abstracts from the set of people who have, or might have, called the elevator by pressing that button.

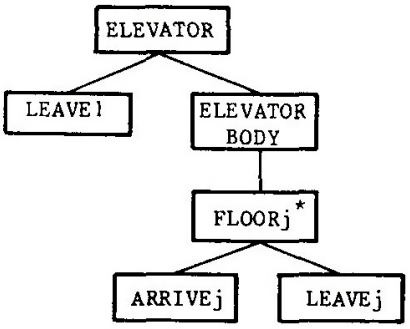
The actions of the button entity are very simple there is only one type of action, namely "be pressed", which, for convenience, we will call PRESS. The actions of the elevator are also simple: they are only LEAVE FLOORj and ARRIVE AT FLOORj. We might have wished to include such actions as "start" or "stop", but the sensor equipment provided is too crude to detect these actions.

4.3 Entity Structures

The structure of BUTTON is:



The structure of ELEVATOR is:

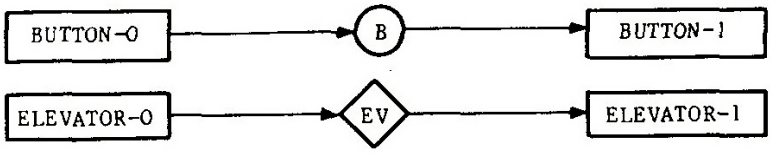


The limitations of the regular grammars we are using prevent us from expressing in the diagram the fact that the successive floor numbers are constrained: $1 \leq j \leq 6$; and, from floor to floor, *j* can change only by *0, +1, or -1*. We expect to use a local variable in the model process, if necessary, to express these constraints. We are assuming, for simplicity, that the elevator begins its life at floor 1, which is the ground floor.

4.4 Initial Model

The elevator engineers have arranged each button so that, when pressed, it sends a signal to the computer. The sensors which detect the positioning of the elevator at the floors, however, are simple make/break switches, whose state can be directly inspected by the computer. The sensor switch at a floor is open when the elevator is not within 6 inches of the home position at that floor, and is closed otherwise. We may regard the set of six sensors as a single variable of 6 bits: at any time, either one or none of the 6 bits is set to 1, indicating that the corresponding sensor switch is closed.

Diagrammatically, we represent the initial model in a System Specification Diagram:



BUTTON-0 is the real button in the real world; it is connected by data stream connection to BUTTON-1, which is the process in the system which models its behaviour. There are sixteen instances of BUTTON-0 and hence of BUTTON-1: 6 within the elevator itself, 5 "up" buttons at floors 1 to 5, and 5 "down" buttons at floors 2 to 6.

ELEVATOR-0 is the real elevator in the real world; it is connected to ELEVATOR-l, which is the process in the system which models its behaviour, by state vector connection. ELEVATOR-1 inspects directly the state of ELEVATOR-0, by examining the value of the set of 6 sensors. The BUTTON-1 process is trivial. It has the same structure as BUTTON-0, and contains operations to read the data stream B. Synchronisation between BUTTON-1 and BUTTON-0 is achieved by these operations BUTTON-1 is blocked at each read B operation until the real BUTTON-0 is next pressed.

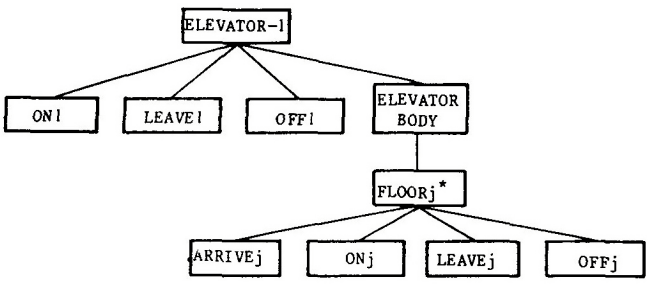
The ELEVATOR-1 process is a little more complicated: it must model the actions of ELEVATOR-0 by repeatedly inspecting the state of ELEVATOR-0 and detecting changes of state from which the occurrence of an action can be inferred. Thus, for example, occurrence of the LEAVEI action is detected when the state changes from "100000" to "000000"; the subsequent ARRIVE2 is detected when the state changes again to "010000", and so on. ELEVATOR-1 is, effectively, in a busy wait loop, waiting for a change of state in ELEVATOR-0. We are using the state vector connection, somewhat reluctantly, to achieve the same effect as would be obtained more easily by the preferable data stream connection.

We may observe that the processes in the initial model are not connected to one another in any way; this is typically, though not universally, the case. The behaviour of each button is certainly independent of the behaviour of other buttons; the behaviour of the elevator will become dependent on the buttons only when we provide some function of the system which will affect the elevator. In this modelling stage, we are concerned only to capture the real world events which we have chosen to model, describing only those constraints which exist separately from the system we are developing.

4.5 Functions: Embedded and Imposed

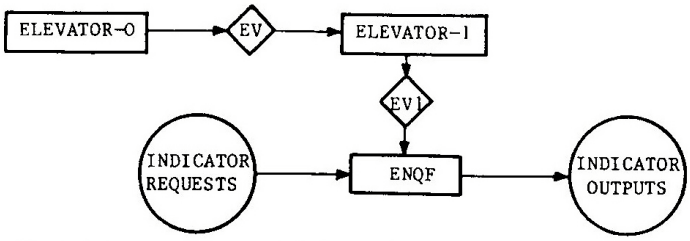
We are now in a position to provide functions which can be specified in terms of BUTTONs being PRESSed and of the ELEVATOR LEAVING and ARRIVING.

A very simple embedded function might be the following. There is a set of lights inside the elevator, one light for each floor. We are required to turn these lights on and off, by issuing outputs ONi and OM respectively for the light associated with floor *i*. When the system begins execution, ON1 must be issued; subsequently, OFFj must be issued when the elevator LEAVEs floor *j*, and ONj when it ARRIVEs at floor *j*. These outputs can be produced by operations embedded directly in the elevator process:



A very simple imposed function might be the following. To cater for nervous or impatient customers waiting at the ground floor for the elevator, a special button is to be provided. When this button is pressed, an indicator beside the button shows which floor the elevator is now at, or, if it is not at any floor, which floor it has most recently visited.

If the ELEVATOR-1 process does not already contain a variable maintained at the required value, we must now provide one. In addition, we must provide a function process ENQF, which will inspect the state of the ELEVATOR-1 process whenever the enquiry button is pressed, and will issue the required output to the indicator:

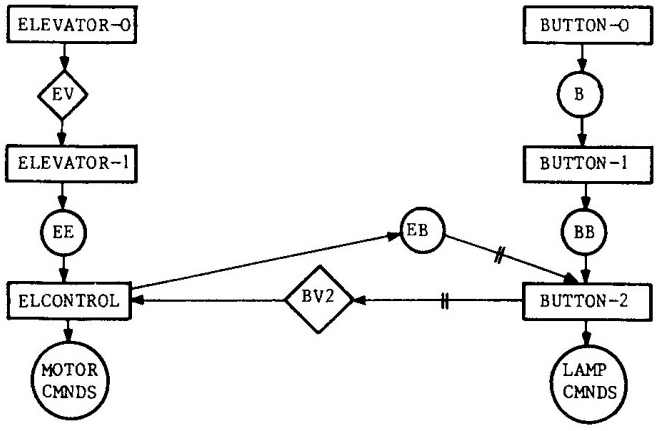


The ENQF process is trivial. It is important to note that it is a process, not a procedure: the lifetime of ENQF spans all the indicator requests that are ever made, not merely one such request.

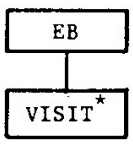
4.6 An Interacting Function

We now turn to the requirement of controlling the elevator. The system must issue commands to the motor to cause the elevator to service user's requests in a reasonable way. The motor commands are GO and STOP, and UP and DOWN, with obvious meanings. In addition, the system must turn on and off the lamp which is beside each button: the lamp is to be turned on when a request is recognised, and off when the request is serviced. The lamp commands are UONi, UOFFi, DONi, DOFFi, EONi, and EOFFi: U, D and E refer respectively to up, down and elevator buttons (i.e., the six buttons inside the elevator).

Clearly, we will need to introduce a function process ELCONTROL to produce the motor commands: the ELEVATOR-1 process does not have the necessary structure. Clearly, also, the ELCONTROL process must inspect the states of the buttons, to determine whether or not a request is outstanding for a particular floor. However, the BUTTON-1 process models only the. PRESS actions, and this is insufficient for the purpose: we need a process which models also the servicing of a request made on the button, that is, one whose state is affected by ELCONTROL itself. This is the classic situation requiring an interacting function: the ELCONTROL process must both inspect the state of a button and change that state. Avoiding the destruction of our model processes BUTTON-1, we introduce BUTTON-2 processes:



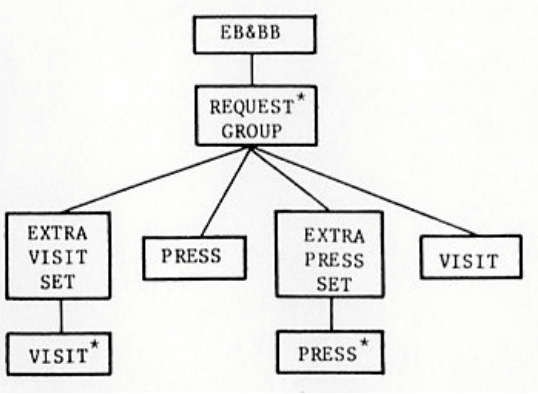
The data streams BB are copies of the streams B. The data streams EB have the structure:



where ELCONTROL writes a VISIT record to a stream EB when it causes the elevator to visit the associated floor in the direction which services requests on the associated button. The data stream EE

may contain records for both LEAVEj and ARRIVEj actions of the ELEVATOR-0, or, perhaps, for ARRIVEj actions only.

The BUTTON-2 process merges its input streams BB and EB. This merge is achieved by BUTTON-2 testing whether a record is available in each stream, and reading accordingly form one or the other. Such a merge depends on the eventual implementation of the system: if the BUTTON-1 process runs faster, or is scheduled more favourably, than the ELCONTROL process, then BB records may overtake EB records; and vice versa. The resulting indeterminacy is an unwelcome, but inevitable feature of the system. The structure of the merged input stream to BUTTON-2, which we will call EB&BB, is then:



which accommodates any degree of "unfairness" in the merge, and also any possible algorithm, however unreasonable, in the ELCONTROL process.

The structure of BUTTON-2 is essentially that of EB&BB; the operations to output lamp commands are easily allocated to this structure. BUTTON-2 must maintain a variable OS, indicating whether a request is outstanding for the button. A request is outstanding after PRESS and before VISIT, and not outstanding after VISIT and before PRESS.

ELCONTROL may have any structure that gives a reasonable servicing of requests. It is important to recognise that we are still developing the specification during the function step: the structure and detail of ELCONTROL will contain the substance of our specification of the control function.

4.7 Implementation

Considering the implementation of the System Specification Diagram shown in section 4.6 above, we need to decide how many processors we will use.

One option is to use one processor for each process. There are 34 processes (16 BUTTON-1, 16 BUTTON-2, ELEVATOR-1, and ELCONTROL), so we would need 34 processors. We would also need a suitable implementation of the data stream and state vector connections, perhaps by use of shared storage locations which are accessible to both the writing and the reading processes of each connection. We can readily convince ourselves that no deadlock is possible even if the implementation provides only one record buffer for each data stream.

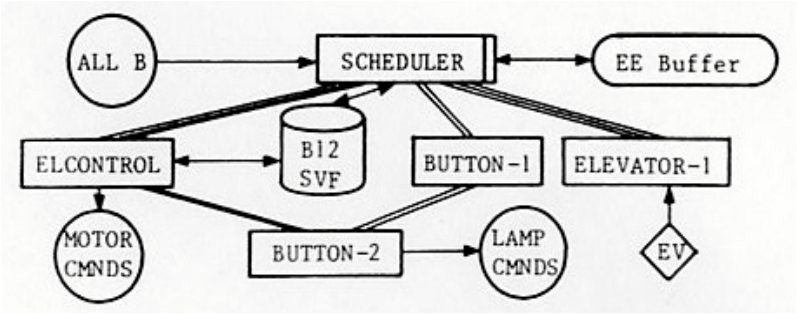
Another option is to simulate the provision of 34 processors by using one hardware processor and a time-slicing supervisor. In JSD we take the view that this is strictly a 34-processor solution. We need to convince ourselves that the time-slices provided to each process give it the equivalent of a processor fast enough to execute the process correctly. For example, the ELEVATOR-1 process must inspect the state of ELEVATOR-0 often enough to ensure that no change of state goes undetected; similarly ELCONTROL must respond sufficiently quickly to the availability of an ARRIVE record in EE to issue the STOP command to the motor in time to halt the elevator at the desired home position.

A more interesting implementation, for our present purposes, is one which uses only one processor: the whole system is arranged so that, from an execution point of view, it is a single sequential process.

A first step in this direction is to consider whether each process contains suitable points in its execution at which it may be suspended by the inversion transformation. Neither ELCONTROL nor ELEVATOR-1 has enough suitable suspension points in this sense. If ELEVATOR-1 is given control, it will enter a "busy wait" loop, inspecting the state of ELEVATOR-0, and will not relinquish control until ELEVATOR-0 changes states. If the elevator is stationary, control will never be relinquished. Similarly, ELCONTROL contains a loop in which it inspects the states of the BUTTON-2 processes, waiting for any request to be made. If ELCONTROL is in this loop, it gill never relinquish control because no

BUTTON-2 process can change state until it receives control. We therefore introduce additional data streams, SE and SC, which are input to ELEVATOR-1 and ELCONTROL respectively; each process reads a record from the additional data stream once in each iteration of its busy loop, and will thus be suspended at the point where the read operation ELCONTROL is inverted with respect to SC and EE; ELEVATOR-I is inverted with respect to SE and EE.

A possible System Implementation Diagram, after introducing these additional data streams, is:



The state vectors of the BUTTON-1 and BUTTON-2 processes have been separated from the executable text, and have been combined into one state vector for each pair, stored in a directly accessible store B12SVF. BUTTON-1 is inverted with respect to its input stream B, and BUTTON-2 with respect to its merged input stream EB&BB.

The merging of the EB and BB streams for each BUTTON-2 process is implemented by making BUTTON-2 a subroutine of both ELCONTROL and BUTTON-I: this is the "fairest" possible implementation of a merge, since the records are read by BUTTON-2 strictly in the order of their writing by BUTTON-1 and ELCONTROL. The buffer for EE is shown, since we do not know *a priori* whether the scheduler will always activate ELCONTROL immediately after ELEVATOR-1 has written a record of EE, nor whether ELCONTROL will always then be suspended at a read EE operation.

The scheduler itself may be quite simple. If a B record is available in its input stream, it may activate the appropriate BUTTON-1 process; otherwise it may activate ELCONTROL and ELEVATOR-1 alternately. This simple scheme may require some modification, depending on the detailed structures of ELCONTROL and ELEVATOR-1, if the processor used has little spare processing capacity: it will then become necessary to consider the execution time for each process activation and, perhaps, to favour one process over another under certain circumstances.

5. Summary

The purpose of this short paper has been to give the flavour, rather than the full detail, of the JSD development method.

The primary advantages claimed for the method are that it structures the development decisions in a sensible way, it leads to systems that can be adapted at reasonable cost when their specifications are changed, and it bridges the uncomfortable gap between specification and implementation.

The small example shown illustrates many of the steps in the method, and some of the relevant considerations at each step. One important point which it does not illustrate is the power of the method to overcome severe mismatching between the dimensions of the specification and the dimensions of the available hardware/software machine. A control system of the kind discussed fits quite well the machine that is, or can be made, available. There are only a few processes, and their demand for machine time is relatively dense. Indeed, there are various operating systems which can execute such a system without discomfort, providing the mechanisms necessary to run such a set of communicating processes.

In a typical data processing system, however, the mismatching would be too great to be overcome by any available, or even conceivable, operating system. Millions of processes, with very long elapsed execution times and very sparse demand for machine time, give a system that is difficult to schedule in an effective manner; the difficulty is compounded by requirements — arbitrary from the point of view of a general-purpose operating system — for particular batching of input and output. The transformations used in JSD then provide the opportunity, which is absolutely essential, of defining explicit scheduling rules for the processes of the specification; and they do so without leading to confusion of specification with implementation.