Proceedings of NordData-80, pages 369-374, Turku, Finland, 27-29 May 1980

System Design for Data Processing

Michael Jackson

1 Introduction

The JSP method of program design bases program structure on the structure of the data to be processed. Each input and output data stream is described in a data structure diagram (essentially a regular grammar); the program structure is a merging of these data structures; the executable operations required to fulfil the program's function are listed and embedded in the program structure at the appropriate places. There is an obvious analogy between this method of program design and the method of compiler writing in which semantic routines are embedded in the parser. Where the data structures of a program are incompatible, a 'structure clash' is said to be present; the program is then structured as two or more sequential processes, communicating by writing and reading intermediate sequential data streams. In this way the structure clash is resolved, and the correspondence between program and data structure is preserved. An implementation technique known as 'program inversion' is used to allow the communicating processes to run in parallel rather than in series: effectively, program inversion transforms a process into a procedure with own variables, the procedure being invoked for each record to be handled by the process. Program inversion may be regarded as the provision of pseudo-coroutines for languages such as COBOL, PL/I and Fortran, with the important difference that it is a transformation technique, rather than a programming language feature.

Extension of this method of program design to system design is based on the adoption of certain underlying principles. Firstly, the fundamental building blocks are seen to be processes rather than procedures; procedures appear as the result of bottom-up design, providing pseudo-elementary executable operations not available in the programming language, or as the result of program inversion. Secondly, design is seen to start from a model of reality rather than from a statement of the program's in ended function: the data structures are described and formed into a program structure before the executable operations are listed and embedded in the program structure. Thirdly, a clear distinction is made between the early stages in which design proceeds in terms of sequential processes communicating by data streams and the later stages in which processes may be transformed into procedures communicating by passing records as parameters.

2 System Design Stages

The distinction between a program and a system is not easy to define; any proposed definition is liable to refutation by counter-example. However, an important characteristic of most significant systems is that the boundaries of the reality with which they are concerned are initially uncertain. The first stage of system design sketches these boundaries by enumerating the entities with which the system will be concerned. For example, a lending library system will be concerned with customers, with books, and so on. Determining the entities at this stage has much in common with the activities of data-base design. However, no connections among entities are yet considered, and no entity attributes. Further, the system function plays no direct role at this stage: the entities considered are solely those which inhabit the reality outside the computer system.

At the second stage, the entity actions are identified, and arranged in time-ordered structures. For example, two actions of a book are 'be loaned' and 'be returned', and perhaps we may also be concerned with 'be rebound'. Clearly, a book cannot be returned before it has been loaned, nor can it be rebound after it has been loaned but before it has been returned. These constraints are expressed in structure diagrams, of the same kind as those used in program design to define data structures. At this stage, the entities are not connected together. It is therefore necessary, for

example, to show both ‘be loaned’ as an action of book and ‘borrow’ as an action of customer. The connection between these actions will be made at the next stage.

We may observe that if a customer of the library is permitted to borrow more than one book at a time, it will be impossible to show in the customer structure the fact that ‘borrow’ and ’return’ actions must be paired. This may lead to the identification of another entity, loan, which had been previously unrecognised as an entity in its own right.

At the third stage, the connections among the entities are established. Regarding each entity as modelled by a sequential process, the connections consist of data streams written by one entity and read by another. Thus, for example, the customer entity, at the point in its structure at which a ‘borrow’ action occurs, may write a ‘borrow’ record to a data stream read by the appropriate book entity; this record must occur in the book entity structure at the point where a ‘be loaned’ action occurs. Communication may be more complex than this. In particular, there may be two-way communication, in which the customer entity writes a ‘borrow request’ record to be read by the book entity, which in turn may write a ‘not available’ record or a ‘loan agreed’ record to be read by the customer.

At the fourth stage, consideration is given to the time grain required in different parts of the system, and also to the functions the system is to perform. Time grain is handled by introducing ‘time grain markers’ into the data structures, both those which define the original sequences of actions by the entities and those which provide inter-entity communication. For example, we might decide in the library system that the significant time grain is ‘day’. We would then elaborate the data structures by introducing markers indicating the end of each day. It would thus become possible to distinguish ordinary loans from overdue loans: an overdue loan is one in which more than, say, twelve time grain markers intervene between the ‘borrow’ and ‘return’ actions.

System functions are provided in two distinct ways. The first, simpler, way, is to embed the executable operation which implements the function directly into the entity processes. For example, we may provide the function of issuing reminders for overdue books by embedding into the loan structure the operation ‘write reminder’ at the point where the loan becomes overdue; similarly, we may arrange to inform a customer that a requested book is not available by embedding the operation ‘write apology’ into the customer structure at the point where a ‘not available’ record is read from the intermediate data stream connecting the customer to the requested book. Naturally, at the implementation stage it will be necessary to collect these outputs together in groups convenient for printing.

The second way of providing a system function does not use embedding of operations. Instead, a separate function process is created, which is privileged to inspect the state of the entity processes directly. Such a function process is using the entity processes as a model whose workings it may examine but cannot interfere with. This relationship between function and model is asymmetric: if process A uses process B as a model, then process B cannot use process A as a model. A hierarchy is thus defined by the relationship; this hierarchy is, in some sense, the most fundamental structure of the system. For example, a ‘library manager’ process might be created, which is privileged to inspect the states of the customers, the loans, and the books, and to determine answers to such questions as ‘how many customers are currently borrowing more than one book?’ or ‘how many books are currently requested but unavailable?’ This library manager process uses the other entity processes as a model, and in this role cannot interfere with their workings but can merely inspect their states. The other entity processes cannot use the library manager process as a model: they cannot inspect its state.

At the fifth stage of design, careful attention is paid to the process scheduling within the system. Where it is necessary to synchronise processes more closely than they are synchronised by the data stream communication, the synchronisation is expressed in the texts of programs which control access to the states of the processes. The library manager may be content with an approximate answer to his queries; but he may prefer an exact answer computed at a particular moment — such as at the end of the day. In the latter case it will be necessary to ensure that the state of each customer is inspected immediately after the customer process has reached the time grain marker for

the day in question; this is the responsibility of the program which controls access to the customer's state.

Broadly, this fifth stage completes the design proper. The design consists of a set of communicating sequential processes, together with the special programs which control access to process states. In principle, it could be run on a suitable machine equipped with suitable software. However, no such machine or software is available in today's usual data processing environment. It is therefore necessary to follow the design activity with a specific implementation activity, in which the system designed is transformed into one which can be run on the available hardware and software.

3 System Implementation

At the heart of the system implementation problem lies the severe mismatch between the processing patterns for which the hardware and software machine has been evolved and the processing patterns of the data processing system. The system has very many processes: in a typical system there will be thousands, tens of thousands, or even millions of processes. Each of these processes runs for a long time: days, years, or even tens of years. Each process spends most of its running time dormant, awaiting the next input record. By contrast, the machine has been evolved to handle programs whose running time is usually measured in minutes or, at most, in hours; it is equipped to handle only a few concurrent processes, each of which spends a large part of its running time in an active state; the machine itself is typically switched off completely each day or each week for a short period; the programming environment copes well with hierarchies of procedures, but less well with networks of processes, especially if they are many in number.

The first, and central, implementation technique is program inversion, the transformation of a process into a procedure which can be invoked once for each record it handles of its input and output data streams. The activation records of the processes become the own variables of the procedures, and are thence separated from the executable procedure text and held as data-base master records. Thus, for example, the activation record of a customer process becomes a customer master record, and so on. Activation of a procedure to handle an input record of one of its connected data streams becomes, essentially:

```
get master record;
invoke procedure (master record, input record);
replace master record;
```

Scheduling of the processes of the designed network can then be chosen at system implementation time, and the choices bound into the texts of specially written scheduling programs. For a transaction-oriented on-line system, a typical scheduling program is a 'transaction-handling module'. When the transaction is input to the system, the module immediately activates the process to which it is directed; if that process writes any output records, the module activates in turn the processes to which they are directed, and so on. By contrast, a typical scheduling program in a batch system is a 'serial file update' program. When transactions are input to the system, they are stored (batched) for some period such as a day or a week; at the end of the period the stored transactions are sorted into the same order as that in Which the activation records are stored for the processes to which the transactions are directed; the update program then activates each process, in this same order, to handle its part of the transaction batch; any outputs written by the processes then form a further batch, to be sorted and treated in the same way, and so on.

Unfortunately, even this special-purpose process scheduling is usually insufficient to produce an efficient system, and further transformations are required. Among these further transformations we may include the use of one process text to schedule directly the activation of a connected process. For example, if the customer process writes a data stream which is read by the book process, we may make the scheduling choice to activate the book process as soon as any record is written in that data stream; activation of the book process as described above (get book master record, invoke book procedure, replace book master record) is then embedded in the customer process in place of the 'write record' operation. The benefit of this transformation is a reduction of local overhead: control is not returned to the scheduling program when the record is written, but instead is passed directly to the book procedure.

Another important transformation is the dismembering of process texts to form convenient 'load modules'. If a 'borrow' transaction is input for a customer, it can be determined at implementation time that only a certain portion of the customer process can be executed as a consequence of this transaction. It is therefore attractive, at least from the point of view of microscopic efficiency, for the special-purpose scheduling program to load into main storage only that portion of the customer process text, rather than all of it. It will be evident that dismembering texts in this way is likely to incur substantial penalties in reduced intelligibility of the implemented system.

In general, it is desirable that transformations should be performed mechanically, since only in this way can we be confident that correctness is preserved. But the choice of transformation must surely remain under the control of the programmer, since it appears that the range of choice is too large for reasonable choices to be made automatically.

4 Conclusion

The design method presented in this paper implies a substantial change in the manner in which the work of creating a data processing system is carried out. The early stages, here referred to as *design*, encompass a large part of what is usually called 'analysis'; at the same time, what is usually called 'design' has been separated into two parts, the first embodied in the early stages and the second treated explicitly as implementation. The early stages constitute a specification phase whose output can then be transformed into the required system. Even in the absence of the highly desirable mechanisation of the transformations, explicit manual transformation at the final stages is perfectly possible; it is surely preferable to the implicit transformation which usually appears as an uncomfortable and confusing ingredient of those early stages which ought properly to be devoted to other things.

As will be evident, much work remains to be done on the design method presented here. It is being carried on by the author and his colleagues, among whom John Cameron should be acknowledged as a major contributor.