Proceedings of Workshop on Software in High-Energy Physics, pages 1-12, CERN, Geneva, 4-6 October 1982

Programming

Michael Jackson
Software Development Consultant
101 Hamilton Terrace
London NW8
England

Abstract

The programmer's task is often taken to be the construction of algorithms, expressed in hierarchical structures of procedures: this view underlies the majority of traditional programming languages, such as Fortran. A different view is appropriate to a wide class of problem, perhaps including some problems in High Energy Physics. The programmer's task is regarded as having three main stages: first, an explicit model is constructed of the reality with which the program is concerned; second, this model is elaborated to produce the required program outputs; third, the resulting program is transformed to run efficiently in the execution environment. The first two stages deal in network structures of sequential processes; only the third is concerned with procedure hierarchies.

1. INTRODUCTION

Conventionally, the written version of a conference paper should omit those introductory remarks of a personal nature — such as compliments and thanks to the Program and Organising Committee — that are appropriate to the oral presentation. But not in this case. In addition to thanking the Committee for inviting me to this Workshop, I must firmly disclaim any knowledge of High Energy Physics and of the programming of problems in that discipline. My unease at this ignorance is somewhat mitigated at finding myself placed so early in the programme of the workshop: clearly, the Committee are anxious to ensure that my paper should be presented in this state of virgin ignorance, before I can have the advantage of hearing other papers and discussions that might impart some fragments of knowledge. It is my hope, and presumably theirs also, that ideas derived from other programming contexts may prove to have some relevance to the problems in High Energy Physics which are the subject of our meeting. With that hope, I will proceed.

2. THE DIFFICULTY OF PROGRAMMING

Programming is not an easy task. This became clear quite early in the history of computing, and has become even clearer as programs become more ambitious in their specifications and in the applications that they serve. Embedded systems and process control systems must be very reliable; computer operating systems are inevitably very complex, and must be efficient; compilers for modern programming languages such as Ada are also complex, and must also be efficient; data processing systems are extremely complex — although their complexity sometimes passes unnoticed because they tend to be loosely textured — and must be capable of constant change as the needs of their users change. All this is well known.

The problem of programming is exacerbated by the material of which programs are built. A bare computer provides a repertoire of simple operations on small data objects: operations to add integers or floating-point numbers; operations to read or write records from or to the external devices, such as disks, tape drives, and line printers; operations to transfer execution control to specified addresses in the stored program. Some means is evidently needed for introducing larger structures into the programmer's mental view of the program. A large program may contain 10,000, or 100,000, or even 1,000,000 machine operations; some organising principle is necessary to deal with a whole composed of so many parts.

The first such principle was established early in the history of programming. Wilkes [14] writes:

“From the very first, I had seen the establishment of a library of subroutines as being of prime importance. Not only would the availability of such a library to draw on save the programmer effort, but library subroutines could ... enable the programmer to work at a level above that of a raw binary computer. The importance of a library of tested subroutines took on a new aspect once practical experience had begun to show how difficult it was to write correct programs. Finally, there was the invention by David J Wheeler of the closed subroutine, which made possible the development of a coherent system of programming based on nested subroutines.”

These two ideas dominated discussion of programming methods for many years, and are still dominant in some circles. The subroutine, or procedure, provides a means of raising the level of the elementary objects that the programmer must deal with. Instead of dealing only with scalar addition and subtraction, he can deal also with multiplication and division (which were often provided by subroutine in early machines); and with exponentiation; he can also deal directly with operations on larger data objects, such as vectors and matrices. Further, one subroutine may contain calls of other subroutines, allowing the whole program to be structured as a hierarchy of subroutines: the top level of the hierarchy may express the whole problem solution in a very small compass, no matter how large the entire program may be.

The other major structuring principle to emerge early was concerned with control flow. The bare machine provided jump instructions, by which control could be passed from any point in the program to any other. It soon became clear that some discipline was needed here also, and broad agreement was reached that control flow constructs could fruitfully be limited to three: sequence, or concatenation; iteration, or repetition; and selection, or choice among alternatives.

Fortran, COBOL, and Algol 60 could be said to have exploited these two principles of structuring: nested subroutines and disciplined control flow. The exploitation, at least in the cases of Fortran and COBOL, was grossly imperfect: both languages have serious anomalies in their control flow constructs, and both have serious limitations on the power of their subroutine calls. But the trend was clear, and all three languages represented a vital step forward in simplifying the programming task.

3. PROGRAM DESIGN WITH SUBROUTINES

In the middle 1960s it was widely agreed that programs should be constructed as hierarchies of subroutines. A design question immediately arose: how should the programmer choose and develop the appropriate hierarchy for each particular problem? How could the subroutine hierarchy be used as a tool in proceeding from the statement of the problem to the finished program? It seemed that the programmer might proceed in either of two directions, or possibly in some combination of them: either from the top of the hierarchy downwards, or from the bottom up.

The top-down approach, in the academically more respectable form known as stepwise refinement, was convincingly advocated in Dijkstra’s influential paper Notes on Structured Programming. The essential idea is that the programmer begins by stating the program in an extremely simple form that uses instructions that are not available in the existing repertoire: this is the top level of the hierarchy. Each of these instructions must then be refined into instructions at the next level, and the development proceeds in this way until the lowest level is reached, where the instructions used are all in the available repertoire. Dijkstra would not, I think, advocate this approach today, but it is widely advocated and used, especially among some groups of people concerned with data processing.

The bottom-up approach has, generally, received less attention. Here, the essential idea is to raise the level of the available instruction repertoire, starting with the level of the programming language to be used. As each level of subroutine is defined, it can be used in the next higher level, until eventually the complete program can be stated in terms of a few high-level instructions. Although this bottom-up approach, in the primitive form described here, has not found much favour, it is clearly discernible in much of the recent and current work on abstract data types[8].

Neither top-down nor bottom-up design is easy to practise. Both may be criticised for demanding too much foresight from the practitioner. Indeed, it is not unfair to say that both are applicable only to problems whose solutions are already known: the programmer first achieves, by hidden intuition,

an outline conception of the whole solution; a top-down or bottom-up approach can then be used to describe this solution in detail. The truth of this criticism can be seen by considering the case of the top-down programmer confronted by several alternative top levels or first refinement steps. How is the choice to be made among the alternatives? Only by exercising a degree of foresight, by recognising that this alternative will lead to greater difficulty than that alternative, at lower levels of the development.

A deeper criticism of both approaches is that they assume a hierarchical structure. The critic who castigates top-down will inevitably be confronted with the reply 'so you prefer bottom-up?': the assumption is that structure is hierarchical, and that the only plausible ways of developing structure are to explore the hierarchy in one of the two obvious orders. The justification for this assumption is partly historical: if subroutines are the sole medium of structure, then structures must necessarily be hierarchical. But it is also related to the Von Neumann machine architecture. A single, bare, Von Neumann processor can execute a program only if it is constructed as a single sequential process, at least as seen by the machine. A hierarchy of closed subroutines naturally forms a single sequential process from the machine's point of view. If the structure of the program as executed must be the same as the structure of the program as designed, it follows that the designer should naturally think of the program as a hierarchy. We will return later in this paper to the relationship between the designed and executed structures.

4. THE PROGRAM AND THE REALITY

Any useful program is concerned to compute about some external reality. A payroll system is concerned with the organisation's employees and the work they do; a program to control a chemical plant is concerned with the vessels and pipes and valves of the plant, and their behaviour. A compiler is concerned with the source language program and the object program that is to be constructed.

In every case, there are good reasons why the structure of the program, in some wide sense, should correspond closely to the structure of the reality with which it is concerned. One reason is that there will then be only one structure to be considered by the programmer: the structure of the problem and the structure of the solution are identical. Another reason is well known in data processing: the need for program maintenance - that is, for changing the program to reflect changed problem requirements. A common source of user dissatisfaction in data processing is the difficulty of program maintenance. The user requests a change that in terms of the problem seems to be small, simple, and local; the programmer examines the program, and finds that in program terms the change is large, complex, and diffuse. This is prima facie evidence that the structure of the program is radically different from the structure of the problem. The resulting high cost and difficulty of adapting data processing systems and program to the constantly changing needs of the user organisation is the theme of countless complaints.

I believe that program development, and system development also, must begin with an explicit stage in which the developer constructs a formal model of the reality with which the program is concerned. Only in this way can the program be given an appropriate structure. It is important to recognise that any program or system will inevitably embody a model of reality: the only question is whether the model is to be explicitly defined or not. The point may be illustrated by a very simple program. This program was shown at a conference on structured programming as an example of approved top-down design; but in fact it shows clearly the inadequacies of that approach.

The problem is to merge two arrays, A and B. Each array element contains an integer key and an alphanumeric value, arranged in ascending index order by integer key. Since the number of elements in each array may vary, there is a special terminating element in each array, having a distinct high key. An example of a pair of arrays is:

```
A: la 2r 2f 4q 5t XX -- --
B: 1w if 2k 3t 3c 5y 6z XX
```

XX is the special terminating element. As shown in the example, there may be duplicate occurrences of a key between the two arrays and also within one array. The rules for merging are that the result array, C, must contain no duplicates: where duplicates exist, the C element must be taken from A in

preference to B, and an element with a lower index in preference to one with a higher. The program presented was essentially as follows, at the second or third step in development:

```
P: initialise indices;
   do while (more to come)
      do while (next C element should be from A)
        if (not duplicate in C)
           transfer A element;
        endif
      increase A index;
      enddo
      do while (next C element should be from B)
         if (not duplicate in C)
            transfer B element;
         endif
         increase B index;
      enddo
   enddo
   transfer XX marker element to C;
end P
```

With suitable further refinement, it can be shown that this program will produce the required array C:

```
C: la 2r 3t 4q 5t 6z XX
```

However, our concern is with the structure of the program: the fact that it works is not evidence of correct structure. The gross structure of the program is:

```
P: initialise;
   do while (..)
      AGROUP: do while (..)
         AELEMENT;
      enddo
      BGROUP: do while (..)
         BELEMENT;
      enddo
   enddo
   terminate
end P
```

AGROUP and BGROUP respectively transfer a group of elements from A and B to C. We may ask ourselves what defines an AGROUP or a BGROUP? For the example arrays given, we can see that the groups are these:

```
1st AGROUP: la     1st BGROUP: lw if
2nd AGROUP: 2r 2f  2nd BGROUP: 2k 3t 3c
3rd AGROUP: 4q 5t  3rd BGROUP: 5y 6z
```

We may say, therefore, that the program imposes a structure on the arrays which groups their elements as shown above. As soon as this is made explicit, it is clear that the structure is wrong: no conceivable view of the problem would be based on these groupings. The penalty paid is that the program would be very difficult to change if, for example, it were required to diagnose the occurrence of duplicate elements, either within one array or between the two arrays. We may also observe that if the elements were placed in the high, rather than in the low, index positions of the three arrays, the grouping of elements would be different from that shown above. Essentially, the structure imposed on the data is an implicit by-product of the algorithm chosen. I am advocating that it should be an explicit product of the first development stage.

5. ONE REALITY AND ANOTHER

In the array example, we have taken the reality to be the data on which the program operates directly. The input to the program is a pair of arrays, and the output is another array. The nature of

the problem invites a treatment of the arrays as sequential data objects, each accessed in ascending order of index.

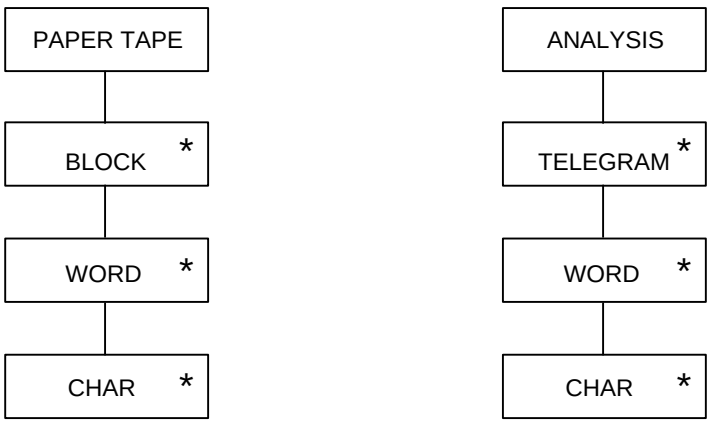
Sequential data objects are found in a wide variety of problem. More properly, we may say that objects are found which can usefully be treated as sequential. Obvious examples include a source program input to a compiler, a stream of input messages at a keyboard terminal, a tape file, and a sequentially accessed disk file. From the programmer's point of view, some of these are inescapably sequential, such as a stream of terminal input messages; others can conveniently be treated as sequential, even if there is no inescapable sequential constraint on their accessing.

Sequentiality is a central aspect of most of those problems whose solutions merit the name of 'system' rather than 'program'. In particular, most systems for data processing, for process controls and switching, and for embedded applications, are concerned with a reality in which time sequencing is a central feature. For a payroll system, the employee joins the company before starting work, goes on holiday before returning to work from holiday, clocks on at the beginning of each day and subsequently clocks off at the end, and so on. For a sales system, the customer places the order, then it is allocated, then it is delivered, then it is billed, and so on. Throughout, we are concerned with time-sequenced actions.

One might draw a distinction between systems and programs based on the nature of the reality with which they are concerned. One might say that a system is concerned with the 'real world' of its users, while a program is concerned only with a reality expressed in terms of computer input and output data. In the array merging problem, we did not ask ourselves what 'real world' the arrays described: we took the arrays themselves to be the reality. Sometimes the two realities - that of the user and that of the computer - are related in an interesting way, illustrated by the following problem. (The problem is an adaptation of a problem discussed by Henderson in a paper on structured program design[9] , and further discussed in other authors' writings [12, 7, 11].) For our present purposes a brief outline of the problem and its solution is enough.

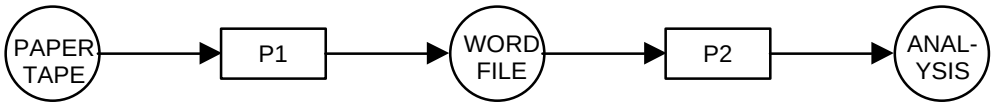
A file of telegrams has been punched into paper tape. The paper tape is arranged in blocks, each block being read into main storage as a variable-length character string, terminated by a special EOB character. Each telegram is terminated by a special word 'ZZZZ', and the whole file is terminated by a telegram consisting only of this special word. The program is to produce an analysis of the telegrams, showing such information as the number of words in each telegram. The difficulty of the program arises from the fact that a telegram may begin or end anywhere in a tape block; one block may contain, for example, the last words of one telegram, another complete telegram, and the first words of a third. Words are never split between blocks.

Following my recommendation that development must begin with an explicit model of reality, we see that the structures of the input and output are, broadly:



The diagrammatic notation means, for example, that the PAPER TAPE consists of some number of BLOCKS, one following another, each BLOCK consists of some number of WORDS, one following another, and so on. (The structures shown are highly simplified, for brevity of exposition.)

We now wish to construct the program so that its structure corresponds to both of these structures, but we cannot do so. Because the boundaries of BLOCKS and the boundaries of TELEGRAMS are not synchronised, we cannot construct a single sequential process reflecting both of the data structures: there is a conflict, or clash, between the data structures [12]. The standard resolution of this 'boundary clash' is to construct the program as two sequential processes, not one. With obvious notation:



WORD FILE is an intermediate sequential stream of records, each of which is a WORD. This solution is dictated by the fact that WORD is the 'highest common factor' between the clashing components BLOCK and TELEGRAM. Process P1 dissects the PAPER TAPE into WORDs, and process P2 builds TELEGRAMs from the WORD FILE. It can be readily demonstrated that the 'structure clash' has thus been removed, and that each of the processes P1 and P2 is trivially easy to design and build. Communication between P1 and P2 is limited to their execution of 'write WORD FILE' and 'read WORD FILE' operations respectively.

We have derived our decomposition of the problem into a pair of sequential processes, communicating by writing and reading a sequential data stream, from a consideration of the structures, or grammars, of the problem input and output data streams. But we might have arrived at the same point by considering the reality lying behind the programming problem. Simplifying a little, we may say that there are two independent actors in the real world: a telegraph clerk, who receives the texts of successive telegrams at a window in the telegraph office; and a paper-tape punch operator, who punches the texts into paper tape. These two actors are independent, except that they are constrained by the rule that the punch operator must punch the words of the telegrams, in their correct order, into the paper tape. No synchronisation of the two actors is specified, beyond the inevitable restriction that the punch operator cannot run ahead of the clerk.

Broadly, this reality is modelled in the two-process solution. The process P1 models (albeit in reverse) the behaviour of the punch operator, while process P2 models the behaviour of the clerk. The WORD FILE data stream is considered to be an unbounded buffer, modelling the freedom of the punch operator to wait an arbitrary length of time before punching each word, after the clerk has made it available.

6. PROCESS NETWORKS

Where time-sequencing is a central aspect of the real world, it is attractive to develop programs and systems in terms of sequential processes. Within one sequential process text we can capture the total orderings by time; in data stream communication among processes we can capture partial orderings. I believe that it is this property of sequential process networks that underlies the increasing trend away from procedure hierarchy design and towards process network design [1,6,10,13].

The freedom to deal in partial orderings brings with it a new responsibility for the developer. Some means must be found of scheduling several sequential processes on a single sequential machine, of implementing a scheduling algorithm that is chosen more or less consciously, and adapted more or less exactly to the needs of the particular problem. The solution to this scheduling problem may lie anywhere on a wide spectrum. At one end, we may use a machine equipped with a general-purpose operating system capable of running many concurrent processes and of providing message-passing services including buffering in queues. At the other end, we may manipulate or transform the program or system so that it becomes, for execution purposes, a single sequential process in which all scheduling of the original processes has been fully bound.

An example of the second approach, applied to the telegrams analysis problem, would be the use of coroutine communication between the processes P1 and P2. A 'master program' initiates execution of the system by 'calling' P1; when P1 has produced a record of WORD FILE, it 'resumes' P2, which in turn 'resumes' P1 when it has consumed that record and is ready to consume the next. Eventually, when P2 has consumed all the records of WORD FILE, it 'detaches'; control then passes

back to the 'master program', and execution of the system is complete [3]. This approach binds the scheduling of P1 and P2 fully, when the system is built: the algorithm is essentially that each process is suspended and activated once for each record of WORD FILE. A similar algorithm may be implemented by transforming P1 into a subroutine ('produce the next record of WORD FILE') called by P2. Such a transformation can be readily systematised [12] and has been mechanised for COBOL programs in a precompiler available from the author's company. In the context in which the transformation is mechanised, the processes P1 and P2 are designed using 'write WORD FILE' and 'read WORD FILE' operations; the required transformation is specified merely by stating how the WORD FILE, the medium of communication, is to be implemented. Evidently, such a transformation can be applied to more elaborate systems of processes, and other more powerful transformations can also be used to bind process scheduling. In general, a process network can be implemented by (a) transforming the processes themselves, and (b) constructing a special-purpose scheduling process which activates and suspends the processes of the network and provides buffering as needed for communication data streams.

Just as increasing attention is being paid to process networks, so there is increasing work on program transformations of various kinds [2,4]. These transformations are not, to this author's knowledge, presented as a means of binding process scheduling, but many of them are well suited to this purpose.

7. UNDERSTANDING PROGRAMS

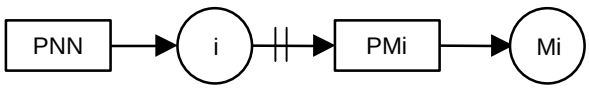
Any development method, whether for systems or for programs, can be seen as a way of structuring and ordering the development decisions that must be taken. The burden of this paper is that for a large class of problem we can recognise three distinct groups of decisions:

- decisions about the reality which furnishes the subject matter of the computation;
- decisions about the program or system outputs containing information about the reality;
- decisions about the scheduling of processes which have been specified in the first two decision groups.

Although all development is necessarily iterative, if only because of human fallibility, we aim to make our decisions in the order given above. A weaker aim, recommended where the first aim is for any reason not adopted, is to understand which group any decision belongs to, and hence to understand more fully what is happening in any actual development work.

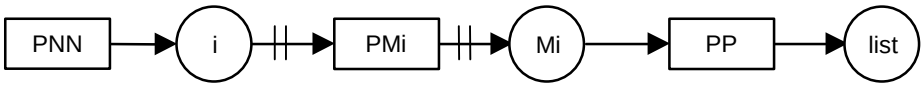
I would like to end by offering another example for the reader to ponder in the light of this stated aim. The problem is the well-known problem of printing the first 1000 prime numbers; a procedure-oriented solution is discussed by Dijkstra in Notes on Structured Programming [5], and a network oriented solution is discussed by Kahn and McQueen [13]. Here no serious solution is attempted: instead some first thoughts are offered, intended to stimulate the reader to consider the problem and the decisions that might lead to a reasonable solution.

We begin by identifying our real world, which is that of the natural numbers. The natural numbers are totally ordered: 0, 1, 2, ... We may therefore construct a process PNN which generates all the natural numbers in order. Each natural number, other than 0, has a totally ordered set of multiples: for examples, the multiples of 3 are 3, 6, 9, ... For each natural number *i*, other than 0, we may construct a process *PMi* which generates all its multiples in order, by execution of addition operations. The processes *PMi* can be created by PNN itself, which writes to each process *PMi* a data stream containing only the single record whose value is the number i. We now have the network



in which the double bar on the arrow entering *PMi* indicates that there are an unknown number of instances of *PMi* connected to PNN. Each data stream *Mi* contains the ordered multiples of the number *i*.

We now turn our attention to the required output, expecting that our of the real world (in which the natural numbers and their multiples are modelled) will be sufficient to provide a list of the first 1000 primes. What is a prime? A prime is a natural number that is a product of no pair of natural numbers except itself and 1. The problem is therefore to identify and list natural numbers satisfying that specification. These are the numbers *p* which appear only in *M1* and in *Mp*. We can easily add an output process PP which collates the *Mi* streams and prints the required primes:



PP can collate the indeterminate number of Mi streams by relying on the fact that if *j* is a multiple of *k* then *j³k*: for a record *k* in *Ml*, only streams *M1, M2, ... Mk* need be considered.

The defects of this solution are many, and most of them are self-evident. The reader, in considering and removing them, is invited to examine each decision and determine whether it is a change to our model of the real world, a change to the function specification, or a change to the scheduling of the system's processes.

References

[1] G R Andrews; *The Distributed Programming Language SR*; Software Practice & Experience 12, 8, August 1982.

[2] R M Burstall & J Darlington; *A Transformation System for Developing Recursive Programs*; University of Edinburgh DAI Research Report No 19.

[3] O-J Dahl; *Hierarchical Program Structures*, in O-J Dahl, E W Dijkstra & C A R Hoare, Structured Programming, Academic Press 1972.

[4] J Darlington; *A Synthesis of Several Sorting Algorithms*; University of Edinburgh DAI Research Report No 23.

[5] E W Dijkstra; *Notes on Structured Programming*, in 0-J Dahl, E W Dijkstra & C A R Hoare, Structured Programming, Academic Press 1972.

[6] W M Gentleman; *Message Passing Between Sequential Processes: the Reply Primitive and the Administrator Concept*; Software Practice & Experience 11, 5, May 1981.

[7] S Gerhart & L Yelowitz; *Observations of Fallibility in Applications of Modern Programming Methodologies*; IEEE Trans SE-2, 3, September 1976.

[8] J V Guttag, E Horowitz & D R Musser; *The Design of Data Type Specifications*; in Current Trends in Programming Methodology Vol IV, ed R T Yen, Prentice-Hall 1978.

[9] P Henderson & R Snowdon; *An Experiment in Structured Programming*; BIT 12, 1972.

[10] C A R Hoare; *Communicating Sequential Processes*; Comm ACM 21, 8, 1978.

[11] W M McKeeman; *Respecifying the Telegram Problem*; University of Toronto CSRG Paper, November 1976.

[12] M A Jackson; *Principles of Program Design*; Academic Press 1975.

[13)] G Kahn & D McQueen; *Coroutines and Networks of Parallel Processes*; IRIA Rapport de Recherche No 202, November 1976 (in English).

[14] M V Wilkes; *Early Programming Developments in Cambridge*, in N Metropolis, J Howlett & G-C Rota, A History of Computing in the Twentieth Century, Academic Press 1980.