Experience of program development with FACT

By O. S. Lumb*

This paper was presented to the B.C.S. Symposium on *Practical Experience with Commercial Autocodes* held in London on 25 March 1964.

Our Honeywell 800 was ordered on 21 March 1963. During the following fortnight all our programmers attended a FACT programming course which was held on our premises. Subsequently, three of these programmers took an Argus (Symbolic machine language) course. We had a number of small jobs, currently done on punched-card equipment, which had to be taken over temporarily until they are absorbed at a later stage. These jobs were used for programmer training, and some of the compilation and debugging was carried out on the Honeywell 800 in London. Our machine was delivered on 30 August and handed over after satisfactory acceptance trials three weeks later. Since handover up-time has averaged 98% and parallel running has averaged 10%. We expect the latter to increase when more work is operational. The compilers and other software have functioned equally well, and comparatively few troubles, except in becoming acquainted with an enormous selection of systems facilities, have been experienced.

Adequacy of programming facilities provided

We have found FACT to be a very comprehensive, powerful and well-documented language. In addition to procedure and file manipulation statements it includes input editors for cards and paper tape, update packages, a report writer, and sort and collate routines.

It was issued in 1961 and the version supplied to us in September 1963 contained many revisions and is a substantial improvement over earlier versions. The current FACT tape still contains five bugs.

Generation of object programs proceeds in three stages:

(*a*) Compilation proper into Argus language. This is carried out on one program at a time.

(*b*) Assembly of Argus into relocatable binary coding. This is a batch process and can be carried out on as many programs as are on the input to assembly tape, at the same time, with little appreciable difference in machine time.

(*c*) Selection of programs, relocation and scheduling for production running.

Alterations can be made to programs at almost all stages, and this facility is used in debugging.

A comprehensive diagnostic system is extremely useful both in detecting language usage errors and in terminating compilation at an early stage if the errors are fatal. In such cases the Console Typewriter types out the Word "Dirty." All FACT statements and the diagnostics are printed out on the printer, and the compilation is abandoned. If the program has no fatal diagnostics it will be compiled ready for assembly on to a symbolic program tape, and a printout of the FACT program is produced automatically. During assembly further diagnostics may be generated, although this is not usual, and an EDIT tape is produced from which a list of the Argus instructions and numerous remarks and comments can be printed if required. This is a most valuable facility for debugging, and although not always necessary we have made a practice of printing this out on every occasion. We shall be more selective in future as experience is gained and machine time becomes less available.

A Program Test System and several other systems programs are available to aid in program testing, of which one in particular, called THOR, is very valuable. This is a general tape-handling and correction program which permits positioning, copying, correcting, editing, rewinding, searching, sampling and comparing of tapes. This system has enabled us virtually to eliminate the need to write programs to test the validity of data on magnetic tapes, although in the beginning several such programs were written and developed, and then found unnecessary. The programming facilities provided are certainly adequate; the problem is that no one person can be expected to know them all, and there are so many "knobs to twiddle" that great care is necessary to avoid misuse and consequential excessive use of machine time.

The following figures have been derived from our records over the past nine months.

Average program size: 400 Descriptor cards.

Average coding generated: 5,100 Instructions + remarks and notes which appear only in the listings.

Time for compilation: 4–5 minutes per 100 cards if program is clean. 1 minute per 100 cards if program is "dirty", i.e. has fatal diagnostic.

Assembly time: The basic time for this operation is about 20 minutes, after which there is a marginal increase for each extra program.

Collection and Scheduling time: 20 to 30 minutes per run.

* *The General Electric Company Ltd., Witton, Birmingham* 6.

The accuracy of the programs produced

The accuracy of conversion from correct source language to machine language is very high, and apart from the cases of known compiler bugs we have had no occasion of faulty conversion due to the software systems.

The language is easy to learn and use correctly, but this does not necessarily mean efficiently. This still depends very much on the individual. Our programmers who took the first course were all new to FACT, but some were fluent in Nebula. They were all able to write programs within a few weeks, but took from 8 to 12 weeks to achieve reasonable competence, and there has been progressive improvement since.

It has been found that previous knowledge of Nebula has been a disadvantage due to the similarity of the two languages causing persistent confusion. We have also found that people with little or no programming experience who have joined the group later, have learned much more rapidly, probably due to the greater experience all round them. As an example, one young lady, with "A" level maths. and no previous experience, who joined the group in July has written and debugged 27 programs, mainly concerned with reports and test programs, which have generated over 150,000 instructions; and she has not had a formal programming course.

Programs are readily intelligible to other programmers, documentation is simplified and systems analysts can be expected to program and implement their work. We have found that a Systems Analyst/Programmer can handle several programs at once and, with experience, the average time to write and debug a program is about two man-weeks. The elapsed time, is, of course, dependent upon how many programs he is handling, and upon getting access to the computer.

However, it is quite clear to us now that the provision of a powerful autocode and compiler cannot relieve the programmer of the obligation to specify exactly and unambiguously, down to the finest detail, the processing to be performed. We have been rather disappointed to find that errors in source language, even after checking by another programmer, have been considerable and form a substantial part of our "bugs".

We are sure that the old idea of having clerical or relatively low grade staff to program in autocode is a fallacy. Detailed systems analysis and programming is a skilled business, and a good proportion of the team must be of the calibre of machine-language programmers and be trained in Argus. FACT is a very powerful aid to the skilled programmer, but it is no substitute for programming skill.

During the past eleven months, including the training period, about 80 man-months of program writing and debugging effort has been put in by the team; 215 programs have been written and had at least one compilation clean of first diagnostics. These programs represent over a million Argus instructions, although, to get to the present position, over $3\frac{1}{2}$ million instructions have been generated through repeated compilations. Of these programs 156 are working and 30 were abandoned either because they were too big and had to be split, or because of system changes or, in the case of test programs, they were no longer necessary. The remainder are still in check-out. The working programs alone represent a net output of approximately 430 machine instructions per man-day.

During the first eight months whilst experience was being gained, the influence of Nebula was still felt, and plenty of machine time was available; the average number of accesses to the computer per clean program was approximately 31, whereas in the past three months it has fallen to 21. More recently a suite of small programs written by an experienced programmer has been developed with, on average, 12 accesses per program. Although not strictly correct one might consider that one complete cycle of compilation, assembly and checkout represents four accesses to the computer.

It is a little difficult to be precise or to draw too firm conclusions from our experience to date. The indication is that although the language is easy to learn it takes about six months to become reasonably proficient, and this is considerably influenced by the general level of experience in the group.

Debugging methods adopted

Debugging is carried out by three main processes:

(*a*) Desk checking of systems and programs.
(*b*) Recompilation of programs after testing against test or live data.
(*c*) Reassembly of programs after testing against test or live data.

Desk checking is important and, as machine availability becomes more restricted, will become more so. It is, however, highly fallible and we are not very satisfied with the results so far achieved. Fortunately, until the program is clean of fatal diagnostics the time used in compilation is small, being between 3 and 6 minutes, depending on the size of the program. In the early days we found that on average $3\frac{1}{2}$ runs were necessary; one program actually had 16 runs before it was finally abandoned as too big. After six months the average had dropped to 2·9 runs per program and there has been a steady improvement since. The diagnostics produced at this stage are of great assistance. In the early days most programmers were dismayed at the errors disclosed.

Once the program is clean of diagnostics it will compile and the FACT listing produced is then checked, although, as in the case of preliminary desk checking, this is not very effective. Very occasionally a program may contain an error which would prevent assembly. In such a case the Console Typewriter will type out "N.G. for Assembly" and the reason. After correction a recompilation is necessary.

The program is then assembled and any diagnostics are printed out with the Argus listing. When these

have been cleared the program can be checked out against test or live data.

This is done by pulling the program off the Symbolic Program Tape via a Program Selection Process. Derails are specified at this point. During the checkout, if derails have been specified, a rough output tape is produced containing the state of core store, special registers and tape records. If the program fails for any reason a dump of the data on the rough output tape at the point of failure can be printed. During the checkout a running commentary of progress indicating the loading of each segment is printed out on the Console Typewriter. This is often adequate, with the aid of the Argus listing, to identify troubles, without having to dump the rough output tape. Many bugs can be found by studying the FACT listing, for example, in file outlines, field lengths, input and report descriptors, and obvious logical errors.

Beyond this point, however, we have found that the use of Argus listings in the hands of a skilled Argus programmer is essential, and even where a bug can be found by inspection of the source language it is often much more readily detected in the Argus listing. There is no doubt that we could not have found the compiler bugs, and probably many others, by any other means. For this purpose one man in particular has become a specialist in Argus and assists any of the FACT programmers when they are stuck.

Many bugs can be corrected by patching in Argus either during a reassembly which saves recompilation, or by one-for-one instruction replacement at check-out time, which is done on the Console. This is done by specifying appropriate loader stop instructions through the Program Selection Process. In the latter case the program has to be recompiled, or reassembled subsequently. At the same time the source language is updated. Wrong coding through bugs in the compiler, for which temporary solutions have been found, can only be corrected by patching during reassembly.

We have found that the majority of bugs can be discovered very quickly, and often programs are ready for reassembly or recompilation within an hour or two of check-out. The more difficult bugs, i.e. those taking more than an hour to find, may take up to a day to detect, and there have been one or two exceptions which have taken longer. One we had about four months ago, in a very large program, could not be tracked down by desk methods and eventually had to be solved on the machine by sub-dividing the program until the error was isolated. This took about $4\frac{1}{2}$ hours machine time.

One software bug caused some annoyance. This was a bug in the report writer which was not discovered for some time. The programmers found that they could exploit the system to get a special format in their output. However, when the bug was discovered and corrected the report writer worked as specified, but we had several disgruntled programmers who had to alter their previously "clean" programs.

In the early days, before experience of Argus patching was gained, programs were repeatedly recompiled, but this is very expensive in machine time. We now resist recompilation as much as possible, unless there is no alternative, as in the case of major structural errors.

In general we have found that debugging is not difficult. The Argus listings which include remarks and comments are invaluable, and the Program Test System has been of great use in checking out programs. The Thor system has virtually eliminated the need to write magnetic-tape testing programs. One lesson we have learned is to break our systems down into the smallest possible programs. In the early days we wrote some big comprehensive programs some of which, after many attempts to make them fit into the core store or get them clean, we abandoned and split into two or more. Small programs take much less machine and debugging time, particularly when a simple change becomes necessary, and there is no problem in running them later under the executive system.

Of all the bugs found it is estimated that about half were errors in systems or logic. Of the remainder 55% were casual errors, 30% due to misconceptions and abuse of the compiler, etc., 10% were due to exceeding memory capacity, misuse of hardware, wrong operation or wrong specifications, and the remaining 5% were due to mis-compilations, including bugs in the compiler.

Problems arising during compilation

There have not really been any problems other than those of inexperience in the use of the language and in operating the computer.

The bugs in the compiler have not caused difficulty in compilation, but only in finding out why the compiled programs would not work.

One small problem is that it is not possible to know for certain whether a program is likely to be too big to fit in the core store, until it has been compiled. As we have learned our lesson with big programs this does not trouble us now.

Satisfactory and unsatisfactory features of the object program produced

Satisfactory

We have found that correct object programs can be produced very quickly, and they are efficient to the point that peripheral units normally operate at full speed.

Sorting and updating programs are highly efficient, procedure statements convert with high efficiency, and it is very doubtful if a skilled machine-language programmer could improve on any of these.

Unsatisfactory

It is very easy to use so many facilities that the object program is cumbersome and slow.

File-handling programs written by unskilled programmers may run at less than magnetic-tape speed, and in general, the input-output routines err on the side of over-checking. Discretion is needed in the use of the report writer in particular. We know that redundant or only marginally useful coding is generated

due to the general nature of the routines, but have not yet had time to establish the full extent of this. It is estimated that the average is of the order of 20%.

Development times for projects are longer than had been hoped, and FACT is very heavy on machine time. There are exceptions; some programs have been developed very quickly, but during the learning period progress has been discouraging.

We are very concerned about the cost of making even minor alterations to a program after it is working. Even in the best regulated circles minor systems changes are necessary, and in a dynamic organization where improvements are constantly being sought, changes could well be frequent. FACT enables the changes to be made easily and in a comparatively short elapsed time, but the cost in machine time can be considerable. This is a problem we shall soon have to face.

In conclusion, although we have had a very trying time since last October, we have got over the learning period and are convinced that FACT is a most powerful language, and the debugging facilities are excellent.

We now know the weaknesses and the pitfalls to be avoided and shall plan future jobs accordingly.

We also intend to set up a small team to investigate the unsatisfactory features with a view to improving the efficiency of the object programs.

One point may be of interest, whenever anything goes wrong the first reaction is that it is either the program or the operator at fault—never the machine or the software.

Book review: Computer organization

Workshop on Computer Organization, Edited by ALAN A. BARNUM and MORRIS A. KNAPP, 1964; 242 pages. (London: *Cleaver-Hume Press Ltd.*, 72s.)

This book is largely concerned with the unconventional for its own sake. Eight papers and subsequent discussions have been reproduced from a conference held in October 1962 in Baltimore, Md.

Two main types of machine organization interest the participants. One is the array of active computing elements, each with its own data store, obeying the same instruction stream from a single control unit. This is exemplified by the SOLOMON computer in Chapter 2 (D. L. Slotmick *et al.*), of which a fuller account will be found in *Proceedings of the F.J.C.C.* 1962; also (in Chapter 3) by a form of matrix organization using optical logic with which readers of *The Computer Journal*, July 1963, will be familiar. The main application is to the sort of numerical problem which is naturally represented on a (rectangular) grid of points, at each of which virtually the same calculation has to be performed, using numbers associated with the point, with its neighbours, or with a "common" pool of parameters. There are two sorts of "edge effects" which arise: one from those points representing the physical boundaries of the problem, the other from the limits of the computing grid, on which the problem is represented by successive overlays. To handle such special cases a form of modal control is introduced, which allows the activity of certain elements to be suspended while others follow through a particular instruction sequence. Obviously the fewer "computer edges" the better. SOLOMON has 1,024 computing elements, and on a selection of suitable problems a performance gain of around 100:1 is claimed in comparison with a "conventional computer." It can also be seen that this organization may be applied to any calculation which has to be repeated many times with slowly changing parameters—slowly changing in the sense that not too many elements will be held up waiting for the last one to go a few more times round a loop. Numerical problems can conceivably be found to satisfy this requirement, and a computer system with a modest number of elements built to process them economically. It is interesting to enquire whether similar situations could be found in data-processing work; on the whole the chances of doing so seem slimmer—the transition between "M" and "F," for example, is usually taken rather abruptly.

Thus the second form of organization requires a multiplicity of control elements, each following its own sequence of instructions, sharing access to data in such a way, one hopes, that an effectively determined calculation is performed. The model taken here for such systems is the Iterative Circuit Computer or Holland machine (see the articles by J. H. Holland in *Proceedings of the E.J.C.C.* 1959 and *Proceedings of the W.J.C.C.* 1960 for up-to-date accounts of this class of machines). I am not sure whether they fulfilled Holland's original purpose, but as starting points in computer design they are rather like skyscrapers without lifts, corridors or telephone systems, taken as starting points for office accommodation. Two of the chapters (4 and 5) in the book and much of the discussion are taken up with, yet nearly miss, this point. When the (blue) sky's the limit there is surely no harm in taking the few steps necessary to get a sensibly programmable system.

The remaining papers offer adjuncts to mainstream programming rather than radical departures. Chapter 1 (D. P. Adams) suggests a mechanization of nomograms, with countable bits on a magnetic or photographic recording replacing the numerical scale markings. It is not clear why this is being attempted, since it appears to offer no advantage over conventional approximation and interpolation techniques. Chapter 6 (G. Estrin *et al.*) describes briefly the efforts being made at U.C.L.A. to devise systematic ways of enhancing the performance of a conventional computer by attaching special-purpose computing elements to work in parallel with it. Chapter 7 (G. J. Culler) outlines a Man-Machine Communication system which sets the fingers twitching, if not the brain. Finally, the paper by N. S. Prywes and S. Litwin describes a design for a computer capable of processing data organized in a generalized form of list structure.

On the whole, therefore, a very disappointing contribution to this interesting field. At least some of the participants seem to have felt the same way, and Professor Pasta leads in with some politely searching questions. It is a little surprising to find exactly the same searching 100-word question on p. 173 as on p. 149, and the verbatim style only heaps on the confusion. There are many errors which one would have expected the editors to notice, but I think the prize goes to the peripatetic (yet powerful!) "touring machine" which appears briefly on p. 211.

J. K. ILIFFE.