The development of a large scale mathematical programming system

by JACQUES DE BUCHET
Metra-Sema
Paris, France

The general purpose mathematical programming systems have greatly changed in the last few years and the aim of such codes is no longer to give access to an efficient algorithm but to integrate the solution of linear programming problems into a loop including the generation of the problem, its solution, its post optimal analysis and the edition of an output report of the results. The link between all these operations is done through a so-called control language program.

All these aspects are more or less the same from one code to another but we think that new approaches should be brought into their design, organization, and implementation. The development cost of such programs being very high, usually in the order of 15 to 30 man years, it is necessary to provide facilities in order to:

- offer suitable users and programming languages;
- have a modular design;
- handle memory allocation in a semi-optimal way;
- make modifications and extensions easier; and
- facilitate day to day programmers tasks.

I shall present, in the following pages, the objectives that we had in mind when we began to implement these facilities within the large scale mathematical programming system OPHELIE 2 which we developed for the CDC 6600.

PROGRAMMING LANGUAGES

A comprehensive set of languages used or provided within a code includes:

- matrix generation language to generate matrix rows, columns and coefficients;
- report generation language;
- control language to link various phases during the solution process;
- programming languages used for the expression of the code;
- updating language to correct or modify programs within the code.

Control Language

It is obvious that out of these five different languages, the first three have common features and specific aspects. The specific aspects include the peculiar functions used for instance to build up and edit a line in a report, or to express the relations between input and output streams through a distillation unit. But these functions are all under control of procedures wnich use the familiar concepts of arithmetic, logical or relationnal expressions, branching and conditional statements and local or COMMON variables which exist in most programming languages. All these common features have been included in the control language.

Thus matrix generation language and report generation language make use of the control language plus specific functions.

In fact we have also extended the use of control language even within programs which make up the body of the code itself. In a code as in other programs, it is possible to distinguish two sorts of functions: command and action, which are emphasized in a modular conception of programming. An action takes place within a module, a command executes initializations and then makes decisions and gives control to a module (Figure 1).

Control language can therefore be used in command programs, the action programs being written in various usual programming languages as they make a comprehensive use of all the facilities available within these languages and as their execution time is sizable.

In order to make programming easier we have extended the capabilities of both control and programming languages as will be described below.

The development of a large mathematical programming system is neverending. It is very common that years after its original design, modifications are still included, this is particularly true in the development and debugging phase.

During that period the modules themselves, once they are working satisfactorily are rarely modified, but

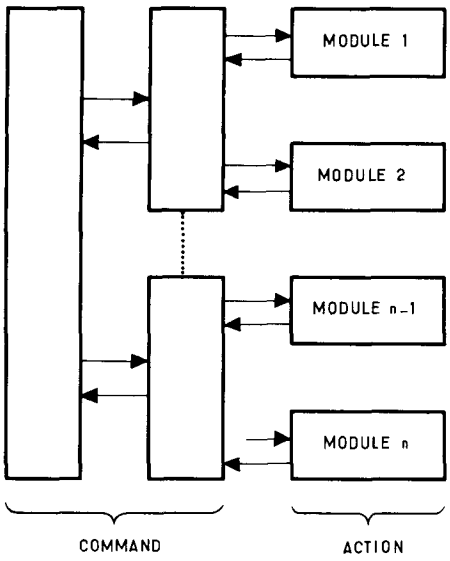

Figure 1—Execution Diagram of Hierarchized Programs

the method of initializing, and executing them changes frequently due to the programmers wanting to replace one module by another, to introduce a check after a module, to insert a printing sequence, etc. . . .

Also, frequently these modifications are only temporary and are kept within the program only for several runs; or it is possible that these modules are effective only for a part of the run, therefore, they need to be introduced just prior to their use, then executed one or several times and removed afterwards.

It is not very convenient to achieve this ability through the standard use of compiled or assembled programs loaded in absolute form in memory. It is much easier if the programs to be modified are kept in symbolic form, that is if we use an interpretative technique. And this is the way the code works. Modules are written in rather standard programming languages, then compiled and loaded in absolute form. Command programs are written in control language and are executed interpretatively.

The control language has the following capabilities:

- has access to a communication region controlling the interface between modules or, between control language programs and modules;
- defines local variables within control language program;
- computes arithmetical and logical expressions;
- branches on simple conditions, or branches unconditionally;
- calls either external programs or internal sequences within the same program; and
- displays the contents of some communication region variables or local variables.

See Appendix 1 for a sample of control language statements.

There are no special difficulties encountered in designing an interpreter for this control language except that control language programs have to be linked with other programs in either a compiled or an interpretative form.

Some difficulties arise with actual and formal parameters due to their type, this problem may be resolved without the necessity of having type declaration statements.

This interpretative technique permits much flexibility and allows dynamic modifications of programs in core, but it is a time consuming technique. Therefore we had a choice to make: use extensively the interpretative form which is flexible but time consuming, or restrict this use to selected command programs which cuts down execution time but also versatility.

As a matter of fact, we chose another possibility. It is possible to distinguish between command programs which are currently in debugging status and in which frequent alterations are made and command programs which work satisfactorily and in which few modifications are made. Let us assume that the former are executed interpretatively and the latter are compiled, loaded then normally executed, provided that there are no modifications to apply, their execution time is minimal. If modifications are to be introduced, we use the interpretative form. Thus the control language is designed to be interpreted or compiled.

It is of course necessary that the compiled and interpreted versions produce exactly the same results, which is not a trivial problem.

Programming language

Command programs have been written in control language, but the main part of the 50,000 symbolic instructions of the code are inserted in modules and it is of course necessary to have access to a well adapted programming language. On a CDC 6600, the choice is restricted to two alternatives: Fortran or macro-assembly languages. Machine language is a must in some time consuming routines such as algorithm or character handling (for instance the interpreter). For all other programs which consume little time we decided to use a higher level language, because of the facility of its use, and also because programs written in this type of language are readily understood by other programmers. This language is basically Fortran plus some modifications designed to increase readibility. We call this language the L language. Each of us has noticed that rapidly looking back into our own programs 6 months, a year, or two years later is not sufficient to understand them, even with the use of comment cards, flow charts and documentation.

The modifications that we have added to Fortran are the following:

- the ability to express variable names and subroutine names with more than 7 characters; our upper limit is 30. With up to 30 characters, it is possible to understand the meaning of any identifier;
- the possibility to give names instead of numbers to a statement identifier;
- the possibility of a dynamic allocation of blocks in core, the programmer no longer being obliged to know where a block is located. For instance, if the solution vector named SOLUTION is situated in a COMMON block named POOL, K words after the beginning of POOL, K varying dynamically, the third component of SOLUTION is in Fortran SOLUTION (3 + K) and in L language SOLUTION (3). It is of course necessary at execution time to handle 3 + K but this is done automatically before compilation;
- an elementary macro capability available at that level to facilitate COMMON definitions. More than 800 programs are used in such a code, most of them use communication region cells or variables in various COMMON blocks. When such a variable is to be referred to in a program, it must be defined within it, and many mistakes are avoided if it is possible to automatically introduce at the beginning of any programs the definition of all or part of the communication region or of COMMON blocks. A modification of the size of a block can be automatically introduced in the 800 programs.

All the specific orders that are available in control language are also available in L language. For instance the possibility to call internal sequences within a program (PERFORM) or to display some information at the console (DISPLAY).

The L language is much too sophisticated to be handled easily by an interpretative routine and used extensively in that mode, it is also quite a considerable task to write a special compiler for it. But in about 6 programmer months it has been possible to design and implement a translator which produces a Fortran version of any program written in that language. As a matter of fact, this translator is also able to generate COMMON and communication region definitions in programs written in macro assembly language.

To help debugging, the translator of L or control language into Fortran produces;

- a source listing in L or control language;
- an object listing in Fortran;
- a list of all the program names and COMMON cell names together with their translation in Fortran;
- for each program a list of the source name for identifiers and labels together with their corresponding translation.

It is worthwhile to notice that the clarity of expression is sufficient enough to allow in most cases a debugging restricted to L language listings by proper insertion of printing sequences or snaps.

There are two kinds of syntax checkings, by the L translator and by the Fortran compiler, in practice no difficulty arises from the necessity to look at both lists because L statements are not deeply modified when translated into Fortran.

See Appendix 2 for a sample of L language statements.

DYNAMIC MEMORY ALLOCATION

The hardware of the CDC 6600 does not allow any paging or dynamic memory allocation, thus memory allocation must be done exclusively on a software basis. The reasons for which we implemented such a package are as follows:

- modularity can be achieved only if modules are independent of the position of the blocks and files in central memory or in auxiliary storage;
- simplicity of programming can be improved drastically if the programmer no longer is required to take care of the file handling and of the use of the same place in core for different blocks and file records;
- efficient use of all central memory can be achieved better automatically by a unique special program rather than under command of the programmer. It is then possible to adapt more easily the size of the blocks or the number of file records in central memory to the size of the linear program, to the memory requirements of the phase under process, or to the central core field length allocated to the job when running in a multiprogramming environment.

In a linear programming code written in a standard form, 20 per cent of the instructions deal with memory allocation problems or I/O optimization.

Before going into further details concerning the memory allocation scheme, some details about the organization of central and auxiliary memories in the CDC 6600 must be given.

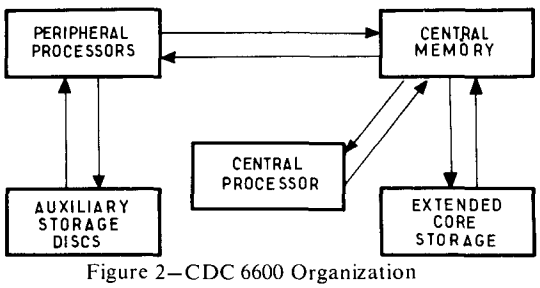

Figure 2—CDC 6600 Organization

The CDC 6600 has 11 processors. One, the central processor which has access to a central core memory and ten peripheral processors which have access to the central memory and to the disc units. The very high transfer rate which is necessary to keep the central processor busy during an algorithmic routine cannot be achieved by the use of a disc. Only the use of extended core storage offers this transfer rate.

Thus, the memory allocation scheme should use a hierarchy of auxiliary devices. All that cannot be kept in central or extended core memory must be stored on discs.

Description of the memory allocation scheme

The allocation of programs in core is not effected by the scheme which deals only with data. Two sorts of data are used in a code: blocks which must be entirely in core when needed and files which are split into records, only one record being active at a time. Scalars or blocks local to a program are not handled by the scheme. All the blocks and files are considered to be in the data pool and each of them is accessed through a call in the communication region which gives its address when in core or an illegal address if out of central core.

Several steps corresponding to a specific treatment such as linear programming data input or optimization or output occur in a typical job. Each of these steps needs to have a particular memory structure. Within a step it is possible to have logical phases. Blocks and files needed are not the same from one phase to another. But, in the same phase all the blocks used must stay in core from the beginning to the end and all the files must have buffers at the same time.

Usually, all blocks and files that are needed in a step are known beforehand as well as the various phase numbers in which they are used. It is then possible to declare at the beginning of a step which blocks and files are needed and in which phases.

Usually, the sequence in which phases are executed is known beforehand, thus it is conceptually possible to allocate room in core for all blocks used in a phase and at least some records for all the files. Depending upon the size of the pool and upon the size of the part allowed for blocks, it is possible to find a feasible solution to this allocation and even to choose the best between several alternatives.

At the beginning of a step, the pool is divided into two consecutive parts, one allocated to blocks and the other to files. The method of handling blocks and files in their respective parts is completely different.

Block allocation

In a mathematical programming system blocks are not necessarily of the same size but can fall into three categories. One in which the block length is equal to the number of constraints in the problem, the percentage of these blocks in 50-70 per cent; another category which is a fraction of the number of constraints plus variables 10-30 per cent; and a third category in which are miscellaneous blocks 10-20 per cent.

We kept the idea of handling three classes of blocks two for the same length blocks and one for the remaining ones.

The principle of the allocation is the following: affect address in core to all the blocks so that at least all the blocks needed in each phase are together in memory at least during that phase and so that from one phase to another the swapping between extended core storage or discs and central core is minimum. The resolution of the problem is of a mixed integer nature and of course it would be too expensive to solve it at the beginning of each step. To find a satisfactory solution, we use heuristics which sound reasonable.

A. Determine the category of all blocks of the step, L1 length, L2 length, miscellaneous.
B. Examine the maximum number of L1 blocks needed in a phase. If we allocate throughout the steps that number to the L1 blocks they can be handled independently of the other blocks.
C. It is then possible that all remaining blocks can stay in core throughout the step. It is also possible that the remaining core memory is not sufficient to handle the other blocks, if this is the case - apply procedure B) to L2 blocks set.
D. Same as C) miscellaneous blocks can stay in core throughout the step or it is not possible to find room for them or it is possible but the miscellaneous blocks must share common places.

We have divided the various blocks into categories which can be handled separately, and at the beginning of each phase blocks must eventually be swapped between main and auxiliary storage. To minimize these transfers, taking into account frequency and sequence of use is desirable. Beginning with the L1 set, we look for a block which is the most frequently needed and try to force it to stay in core during the entire step, then we determine if it is possible to handle the other blocks of the set, in that case we do the same for all the blocks of the set in a decreasing priority order up to a moment where a block cannot be forced to stay in core throughout the step, in this case we skip this block and do the same for a block of inferior priority. The priority being a function of frequency of utilization and for the same frequency the smallest blocks are given priority. The same algorithm is applied to L2 and to miscellaneous block sets.

File allocation

During a phase each file must have at least some buffers in core. It is very often desirable to have various numbers of buffer for each file according to its use. To define the fule buffers allocation scheme, we assumed that a file which is in a reading state during a phase is entirely scanned forward or backward, that it is possible to write files or to read a file and add records to it. Contrary to the block case, the length of the files is not known at the beginning of the step therefore it has not been possible to use the same priority criterion, but as the length of all the records of all the files is the same it has been possible to affect priority to files at the beginning of a step and to adapt dynamically the number of buffers allocated at least for the high priority files. The file which is most often read in all the phases is considered to be the file with the highest priority, a maximum number of buffers is devoted to that file. In fact this number depends upon the file length and is dynamically adapted. The remaining buffers are assigned to other active files of the step.

This allocation scheme seems perhaps far from being optimal, but it should be kept in mind that dynamic allocation must represent only a small percentage of the total time, 10 per cent at maximum, and that two factors make things easier: first the mathematical programming system runs in a multiprogramming environment so that control can be given to another person during I/O requests. And secondly that transfers between extended core storage and central memory not being, simultaneous, it is not necessary to issue read or write requests to or from ECS in advance. In any case, we keep track of all the transfers which are made and we will have some figures which can help to adjust or modify the basic algorithms.

We believe that a well designed code should not only make its implementation or implementation of extensions easier but should also facilitate the programmers task. For instance, using well adapted languages to write the code, having a satisfactory modularity or a dynamic memory allocation scheme is not all that is desirable. The conditions in which runs are made are also very important. Before each run, corrections are to be made to the symbolic text of the code, followed by compilation or assembly and loading, and permanent corrections should be dispatched to every programmer. A suitable design of the updating system can save much clerical work and errors.

During the debugging period or later on when additions are made, it is very convenient to have an extensive list of all the programs which use a specific variable, usually a communication region cell, this is possible through the translator which was described earlier.

We believe that permitting a lot of facilities within the development phase is a rewarding operation. It allows easy extensions to the system many years after the original design. This is obviously a necessity because even when the options given to the mathematical programming system users are extensive, most of these users want to develop special features of their own to adapt the code to their specific problems.

Any matrix generator, control language, report generation or mathematical programming algorithm should be easily modifiable and extendable.

APPENDIX 1

SAMPLE OF CONTROL LANGUAGE STATEMENTS

```
          /M/ SUBROUTINE MAIN PRØGRAM
C                                          EXAMPLE OF CONTRØL LANGUAGE
          CØMMØN /CR/N,NUMBER ØF ROWS,TEXT
          LØGICAL ERRØRS IN DATA
          DISPLAY ≠TØØ MUCH ERRØRS IN DATA≠
          N= N+10 *(NUMBER ØF RØWS-1)
          IF(N.GE.5) ERRØRS IN DATA =.TRUE.

          INDEX=NUMBER ØF RØWS
 *RESET TØ ZERØ*  SØLUTIØN(INDEX)=0
          TALLY INDEX,RESET TØ ZERØ
          MØVE TEXT= ≠TØØ MUCH ERRØRS IN DATA≠
          RETURN
          END
```

APPENDIX 2

SAMPLE OF L LANGUAGE STATEMENTS

```
              /L/SUBRØUTINE CØLUMN HANDLING
              CØMMØN /CR/
              CØMMØN /BLUE/
              CØMMØN /CØLUMN/CØLUMN(2500)
              TYPE DØUBLE CØLUMN
C
C                                          THIS PRØGRAM WRITTEN IN L LANGUAGE
C                                          LØØKS FØR A CØLUMN IN MATRIX FILE,
C                                          THE CØLUMN NUMBER IS GIVEN BY SUBRØU
C                                          TINE FØLLOWING CØLUMN
              IF(RECØRD NØT FINISHED) GØ TØ LØØK IN RECØRD
              RECØRD NØT  FINISHED=.TRUE.
              PERFØRM NEW RECØRD
    *LØØK IN RECØRD*
              INDEX=INDEX +NUMBER ØF CØEFFICIENTS +1
              IF(MATRIX(INDEX).EQ.O) PERFØRM NEW RECØRD
              EXTRACTIØN ØF MIDDLE BITS(MATRIX(INDEX),NUMBER ØF CØEFFICIENTS)
              RETURN
  * NEW RECØRD*  FILE READ(MATRIX, *ERRØR* )
C                                          (LØØK IF VARIABLE IS IN RECØRD)
              NUMBER LAST VECTØR=MATRIX(1).AND.777777B
              NEXT
* ERRØR *     DISPLAY ≠ END ØF FILE ØN MATRIX FILE≠
              RETURN
              END
```