MCL - OPTIMIZATION ORIENTED PROGRAMMING LANGUAGE

C.S. CHASSAPIS, D.G. PAPAGEORGIOU and I.E. LAGARIS

Physics Department, Applied Physics Laboratory, University of Ioannina, P.O. Box, 1186, GR-45 110 Ioannina, Greece

Received 2 June 1988

MCL is an optimization control language associated with the recently published MERLIN optimization package. It is developed so as to aid in constructing effective minimization strategies. It is very simple and easy to master. Supports Fortran-like operations, conditional and unconditional branching, multidimensional arrays, loops, statement functions, I/O operations and offers quite a few intrinsic functions. MERLIN has been further developed too, so that now it can be driven by the object code produced by the MCL compiler.

PROGRAM SUMMARY

Title of program: MCL

Catalogue number: ABHA

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Computer: CDC CYBER-171; *Installation*: University of Ioannina, Ioannina, Greece

Operating system: NOS 2.5.2 678/670

Programming language used: ANSI FORTRAN 77

High speed storage required: 31000 words

Number of bits in a word: 60

Peripherals used: terminal, disc

Number of lines in combined program and test deck: 9907

Separate documentation available: manual (129 pages)

Keywords: programming language, compiler, minimization, data fitting, MERLIN

Nature of physical problem
Many problems in physics, chemistry, applied mathematics as well as in engineering and in other fields, may be reduced to minimizing a function of several variables. MCL is a system designed to enable one to develop systematically efficient minimization strategies.

Method of solution
An optimization control language associated with the recently published MERLIN package [1] is developed, to aid in constructing effective minimization strategies. It is very simple and easy to master. Supports Fortran-like operations, conditional and unconditional branching, multidimensional arrays, loops, statement functions, I/O operations and offers quite a few intrinsic functions. MERLIN has been further developed too [2], so that now it can be driven by the object code produced by the provided MCL compiler.

Restriction of the complexity of the problem
The size of the compiler tables is set so as to suffice for the needs of most programs. However, if needed this can be reset as described in the manual.

Typical running time
Running time heavily depends on the size and complexity of the input program. The provided test run, took 3.1 CPU s on a CDC Cyber-171.

Unusual features of the program
Provision has been taken so that user-extensions to MERLIN-2.0 are supported through a special statement (command EXECUTE).

References
[1] G.A. Evangelakis, J.P. Rizos, I.E. Lagaris and I.N. Demetropoulos, Comput. Phys. Commun. 46 (1987) 401.
[2] D.G. Papageorgiou, C.S. Chassapis and I.E. Lagaris, Comput. Phys. Commun. 52 (1989) 241.

LONG WRITE-UP

1. Introduction

MCL stands for MERLIN Control Language, and is a special purpose programming language. The recently published MERLIN package [1], is a portable optimization program with six minimization algorithms and a friendly operating system. MERLIN accepts commands that determine its route of action. The user may prepare an input file to MERLIN, containing commands that are to be executed sequentially. This however is a static, rather than a dynamic way of using MERLIN. A more efficient way is to use it interactively, entering the commands through a terminal. For example, one may realize during execution, that one of the implemented algorithms has a superior performance and so may decide to use it repeatedly, or (since MERLIN commands are parametric) one may determine the values for the so called Panel-parameters that seem to be most effective. On the other hand, this requires quite some time of on-line experimentation in front of a terminal which is highly undesirable. MCL is built to deal primarily with this problem. It is a comprehensive and easy to learn language that controls MERLIN dynamically, in the same way a user would from a terminal, yet it does not require the user's attention or intervention. MCL supports branching structures, arithmetic operations and other features described in detail in this article. To make this possible, we constructed the MCL-compiler and we added to MERLIN an appropriate "interface" package of subroutines that perform the additional necessary procedures required by the language (see ref. [2]). The MCL object code (MOC) is input to MERLIN and dictates its execution at run-time. The MCL compiler is named MCLCOM 1.0 and the new MERLIN code is referred to as: "MERLIN 2.0 – Enhanced and Programmable Version". This new 2.0 version [2], apart from being programmable, offers to the interactive user as well, some new capabilities. For a better understanding of this article and for immediate use of MCL, one needs to be familiar with the MERLIN optimization system, at the user level.

2. Description of the language

Only floating point arithmetic is supported. Each number must have at least one digit before the decimal period. For example:

0.15 is an acceptable number, while 0.15 is not.

There are three types of operators in MCL. Arithmetic, relational and logical. The arithmetic operators are the usual ones:

+, −, *, / and **

standing for addition, subtraction, multiplication, division and raise to a power. The relational operators are:

>, <, >=, <=, :=: and #

and they are equivalent to the Fortran, .GT., .LT., .GE., .LE., .EQ. and .NE. operators.

The logical operators are: NOT, AND, OR and XOR, standing for the usual logical operations in an obvious notation. The associativity of all of the above operators is from left to right. The rules for evaluating expressions involving the above relational and logical operators are given in table 1. Note that *an MCL expression is considered by convention FALSE if its value is zero and TRUE otherwise.*

Table 1

$$\textbf{NOT}\ \text{expression} = \begin{cases} 1, \text{ if expression} = 0 \\ 0, \text{ otherwise} \end{cases}$$

$$\text{Expr1}\ \textbf{AND}\ \text{Expr2} = \begin{cases} \text{Expr1}, \text{ if Expr2} \neq 0 \\ 0, \text{ otherwise} \end{cases}$$

$$\text{Expr1}\ \textbf{OR}\ \text{Expr2} = \begin{cases} 1, \text{ if Expr2} \neq 0 \\ \text{Expr1}, \text{ otherwise} \end{cases}$$

$$\text{Expr1}\ \textbf{XOR}\ \text{Expr2} = \begin{cases} \text{Expr1}, \text{ if Expr1} \neq 0 \text{ and Expr2} = 0 \\ \text{Expr2}, \text{ if Expr2} \neq 0 \text{ and Expr1} = 0 \\ 0, \text{ otherwise} \end{cases}$$

$$\text{Expr1}\ \textbf{RelOp}\ \text{Expr2} = \begin{cases} 1, \text{ if Expr1 } \textbf{RelOp} \text{ Expr2 is true} \\ 0, \text{ otherwise} \end{cases}$$

RelOp is any of the relational operators:
>, <, <=, >=, :=:

All operators are listed below in order of descending priority:

NOT (highest priority)
* *
*, /
+, −
>, <, >=, <=, :=:, #
AND
OR
XOR (lowest priority)

A valid MCL line can be up to 80 characters long; longer lines are subject to truncation and may lead to inexplicable errors. Line continuation techniques are not supported. Blank lines, as well as leading blanks, are ignored. However, they may be used, so as to improve the readability of a program. MCL supports comment lines. A comment line is specified by the special character >, at the first non-blank position of an MCL line. Comments may also be appended at the end of any MCL line following the percent character %.

Symbolic names are alphanumeric strings of up to 30 characters, the first of which must be a letter. Underscores can be used anywhere in between (even into keywords), for the sake of clarity and are ignored by the compiler. For example, the symbolic name:

THIS_IS_AN_ACCEPTABLE_NAME_BY_MCLCOM

is a valid 30 character long symbolic MCL name, and equivalent to:

THISISANACCEPTABLENAMEBYMCLCOM (underscores do not count).

All MCL keywords can be used as symbolic names with the exception of the following:

JUST, THEN, FROM, BY and TO.

MCL supports arrays, statement functions and offers to the user a number of intrinsic functions and MERLIN variables.

There are two kinds of statements in MCL. Declaration and control statements. In every program the declaration statements must precede the control statements.

2.1. Declaration statements

MCL offers two distinct types of declarations, through the VAR and the FUNCTION declaration statements. VAR declarations must precede FUNCTION declarations.

i) The VAR statement

All variables and arrays used in an MCL program must be declared. An array is a sequence of numbers referenced by one symbolic name. When an array name is qualified by one or more subscripts, depending on the array's dimensionality, it refers to an individual element of the sequence. Subscripts may be any valid expression. An array can be of any dimensionality, the only limit is imposed by the length of the MCL line. For simple variables the declaration is as:

VAR *A*; *B*; *C*;...

where *A*, *B*, *C*,... are symbolic names corresponding to simple (not array) variables used in the program. For arrays the declaration is as:

VAR *W*[*L*1 : *U*1, *L*2 : *U*2,...];...

where *W* is a symbolic name that will be used to reference the elements of the array. *L*1, *U*1, *L*2, *U*2,... are called dimension declarators. The L and U specifiers in a dimension declarator L : U are the lower and upper dimension bounds, respectively, and they must have integer values (negative, positive or zero). Both simple variables and arrays can be declared in a single VAR statement.

Some examples of valid VAR statements are

VAR A;PRINCIPIA;NEW_STUFF[1 : 8, − 1 : 1, − 11 : 100]

VAR VOLUME[− 1 : + 8];ROOT;MINIMUM;PISA

One can have any number of VAR statements in a program, according to his needs. There are a few variables and arrays that can be used in a program without declaration. These are all intrinsic MERLIN-variables used to monitor the optimization process. They should never be declared. Their symbolic names are MCL-reserved keywords and are listed in table 2, along with their significance.

ii) The FUNCTION statement

The FUNCTION declaration statement offers the possibility to reference an expression through a symbolic name. A statement function is defined in the following manner:

FUNCTION *Fname* [*Arg*1, *Arg*2,...] = *Expression*

where *Fname* is the symbolic name used to reference the function, *Arg*1, *Arg*2,... is the dummy argument list and *Expression* is any valid MCL expression using the arguments from the dummy argument list. A statement function must not contain a forward reference to another statement function and its symbolic name should not coincide with any of the reserved keywords or with any other symbolic name previously declared. Array elements and intrinsic variables are not allowed into the dummy argument list of a function. Some examples of valid FUNCTION statements are given below:

FUNCTION G[A,B,C] = A * * B − B * * C

FUNCTION ROOT[A,B,C] = (− B + SQRT[B * * 2 − 4 * A * C])/(2 * A)

FUNCTION F[A,B] = 3. * A − X[3] + B * * VALUE + FFF/2

Since the user may find a bit confusing the use of both commas and semicolons as separators and the use

Table 2

X[1 : DIM]	Current values of the minimization parameters
L[1 : DIM]	Lower bounds for the minimization parameters
R[1 : DIM]	Upper bounds for the minimization parameters
FIX[1 : DIM]	Fix-status for the minimization parameters, zeros correspond to fixed variables
GRAD[1 : DIM]	Partial derivatives of the objective function
MARG[1 : DIM]	Bound-status for the minimization parameters, the values: -1, 1, 2 and 0, indicate the existence of: lower, upper, lower and upper, and no bound, correspondingly
STEP[1 : DIM]	Current values for the search-steps
TCOUNT	Total number of calls to the objective function
PCOUNT	Number of calls to the objective function, counted since the last issue of a RESET statement
VALUE	The current value of the objective function
DIM	The number of the minimization parameters (i.e. the dimensionality of the objective function)
PRECISION	Equals to: 10^{-d}, where d is the estimated number of significant digits used by the machine
BACKUP	Equals 1, 2, 3 for the BACKUP options: NOBACK, LASTBACK, FULLBACK
DERIVA	Equals 1, 2 for the DERIVA options: ANAL, NUMER
PRINTO	Equals 1, 2, 3 for the PRINTO options: NOPRINT, HALFPRINT, FULLPRINT
USERSO	Equals 1, 2 for the USERSO options: ROOKIE, EXPERT
CALLBY	Equals 1, 2 for the CALLBY options: NAME, INDEX
PROCES	Equals 1, 2 for the PROCES options: BATCH, IAF
PANEL	Equals 1, 2 for the PANEL options: PANEL-OFF, PANEL-ON

of both parentheses and square brackets as grouping marks, the following simple rule should be kept in mind:

Array subscripts, as well as function arguments, are separated by commas and grouped by brackets, while statement parameters are separated by semicolons and grouped by parentheses.

MCL offers a number of intrinsic functions whose names are reserved keywords, and need no declaration. They are all listed in table 3 along with their meaning.

When referencing an intrinsic or a statement function, the arguments can be either simple variables or valid MCL expressions.

2.2. Control statements

These are divided in the following categories: Assignment, I/O, Conditional and Unconditional flow-control, Loop and MERLIN statements.

2.2.1. Assignment statement

The assignment statements are used to store a value in a given variable or array element. The syntax is as:

$V =$ *Expression*

where V is the symbolic name of a simple variable or array element, whose contents are to be replaced by *Expression*. We must point, that V cannot be any of the intrinsic MERLIN variables or array elements. For these there are special statements that one should use.

Expression can be any valid MCL expression. A few examples are shown in table 4.

Table 3
Intrinsic functions (all names are reserved)

ABS[X]	$\|X\|$
TRUNC[X]	integer part of X
ROUND[X]	nearest integer to X
SQRT[X]	$\sqrt{X}$
LOG10[X]	$\log_{10}(X)$
LOG[X]	ln(X)
SIN[X]	sin(X)
COS[X]	cos(X)
TAN[X]	tan(X)
ASIN[X]	arcsin(X)
ACOS[X]	arccos(X)
ATAN[X]	arctan(X)
SINH[X]	sinh(X)
COSH[X]	cosh(X)
TANH[X]	tanh(X)
ASINH[X]	arcsinh(X)
ACOSH[X]	arccosh(X)
ATANH[X]	arctanh(X)
MOD$[X_1, X_2]$	X_1 modulo X_2
MIN$[X_1, X_2,\dots,X_N]$	$\min\{X_1, X_2,\dots,X_N\}$
MAX$[X_1, X_2,\dots,X_N]$	$\max\{X_1, X_2,\dots,X_N\}$
MEAN$[X_1, X_2,\dots,X_N]$	$(X_1+X_2+\cdots+X_N)/N$
FACT[X]	factorial of the nearest integer to X

2.2.2. *Input / output statements*

MCL handles its input via the GET statement and its output via the DISPLAY statement.

i) The GET statement

The syntax is either:

GET *A*; *B*;...; *D*

or

GET *A*; *B*;...; *D* **FROM** *FileName*

where *A*, *B*,..., *D* are symbolic names of simple variables or array elements. *FileName* is the name of the file, where the corresponding values reside. For every GET statement, all the corresponding input data must be in a single line.

Table 4

Plain expressions	MCL-expressions
$\text{Tri} = ax^2 + bx + c$	TRI = A * X * * 2 + B * X + C
$z = m + 1/a$	Z = M + 1/A
$a_i = x_i - 1$	A[I] = X[I] − 1
$y = \arctan(x)$	Y = ATAN[X]
$y = \sqrt{x}$	Y = SQRT[X]
$(x < 0)\cup(y \geq 1)$	(X < 0) OR (Y >= 1)
$(x > 0)\cap(y \leq 1)$	(X > 0) AND (Y <= 1)

When the first format is used, MERLIN's default input file is assumed. Upon execution, the contents of A, $B, \ldots, D$ assume the input values, in an one to one correspondence from left to right. A few examples follow:

GET A;I;B[I − 2];PRESSURE;FIRST_ROOT

GET C;D **FROM** SPECIAL_FILE

ii) The DISPLAY statement

The syntax is either:

DISPLAY *A*; *B*;...; *D*

or

DISPLAY *A*; *B*;...; *D* **TO** *FileName*

where A, $B, \ldots, D$ are either expressions or character-strings delimited by single quotes. Character-strings may contain any character except the single quote (′) and can be at most 60 characters long. Each arithmetic expression is evaluated and its value is output. For each delimited character-string the output consists of the string itself. Note that underscores do appear in the output if they are part of a string.

FileName is the name of the file, to which the output is directed. When the first format is used, MERLIN's default output file is assumed. In the examples below

DISPLAY 'FINAL_VALUE IS';FF;'CENTIMETERS'

DISPLAY 'NEW PERIOD';SQRT[4 * X[1]] **TO** FILE1

we obtain the following output in the default output file, and in FILE1, respectively (assuming FF = 13.0 and X[1] = 1).

```
FINAL_VALUE IS 13. CENTIMETERS
NEW PERIOD 2.
```

As you may observe, the output format for numbers is fixed.

It should be kept in mind that the following filenames are *reserved* by MERLIN and cannot be used for I/O:

HELP, DATA, STORE, INIPO, DISPO, BACKUP and **MACROF**.

2.2.3. *Conditional flow-control statements*

(i) The Block-If

A Block-If consists of the IF *condition* THEN, ELSE and ENDIF statements. Each IF *condition* THEN statement must be balanced by an ENDIF statement. A Block-If provides for the selective execution of a particular block depending on the validity of the *condition*. The syntax of the Block-If is:

```
IF condition THEN
   Block of: statements, label-lines, comments
ELSE
   Block of: statements, label-lines, comments
ENDIF
```

with *condition* being any valid MCL expression.

The ELSE statement and the second block are optional. If *condition* is true, the first block is executed and the control transfer to the statement immediately following the ENDIF statement. If *condition* is false, then, if a second block exists, it is executed and the control transfer to the statement immediately following the ENDIF statement.

No jumps are allowed inside an IF-block from a point not belonging to that block (via a MOVE statement described later on).

Each block may contain more than one Block-If constructs. Since each Block-If must be terminated by an ENDIF, there is no ambiguity in the execution path. Note that both ENDIF and END IF are acceptable.

The following is part of an MCL program, that may serve as an example of a Block-If structure:

```
IF (X[K] <= 77.9) AND (VALUE <= 1.0E-3) THEN
   SIMPLEX (NOC = 100 * MEAN[RATE[1],RATE[2]])
   SHORTDIS
ELSE
   ROLL % use ROLL with the default parameters.
   GRADDIS (ERR = 0.00094)
ENDIF % end of Block-If structure.
```

(ii) The WHEN statement

This is a one-line conditional statement. Its syntax is:

WHEN *condition* **JUST** *ncnlcs*

with *ncnlcs* being any control statement besides the LOOP and the conditional statements.

If *condition* is true, then the *ncnlcs* statement that appears immediately after the **JUST** keyword is executed. If the *condition* is false, the control transfers to the statement of the next line and the *ncnlcs* statement are ignored.

As an example one may have:

```
WHEN TRUNC[J] :=: 11 JUST MINIMAL = VALUE
```

2.2.4. Unconditional flow-control statements

These refer to the statements MOVE, FINISH and PAUSE.

The MOVE is the only unconditional branching statement supported by MCL and its syntax format is:

MOVE TO *word*

where *word* is the symbolic name of the label at which the control is to be transferred. A label-line has the form:

word:

where *word* is the label name, and it can be any valid MCL symbolic name. Note that it has to be followed by a colon. Labels are used inside a program as addresses where the control may be transferred. Duplicate label names are not allowed.

The FINISH statement results in program's termination. The control is transferred to MERLIN's operating system. The syntax format is:

FINISH

If an MCL program ends without a FINISH statement, even though this is not aesthetically pleasing, it will not cause any problems, since the compiler places a FINISH at the end of the object code anyway.

Table 5

```
> This program calculates the sum of the ten elements of array A
  VAR I;SUM;A[1:10]
  SUM = 0
  I = 0
  START:
     I = I + 1
>    Read a value for A[I] from the file: MYFILE
     GET A[I] FROM MYFILE
     SUM = SUM + A[I]
>    Check if all ten elements are added.
     WHEN I < 10 JUST MOVE TO START % repeat the procedure
>    Output; the message: SUM =,
>    and the value of the variable SUM, to file RESULT.
     DISPLAY'SUM =';SUM TO RESULT
>    Terminate program's execution:
  FINISH % Control is transfered to MERLIN.
```

The PAUSE statement results in execution's suspension. This command is only for interactive MCL-use. One can resume execution with an entry from the keyboard. The syntax format is:

PAUSE

An example illustrating the use fo MCL features described so far, is shown in table 5.

2.2.5. Loop structure

A) The LOOP statement

This statement provides the means for repeating a block of statements. Its syntax is:

LOOP *Var* **FROM** *init* **TO** *final* **BY** *step*
  Block of statements, label-lines or comments
ENDLOOP
  or
LOOP *Var* **FROM** *init* **TO** *final*
  Block of statements, label-lines or comments
ENDLOOP

where *Var* is a simple variable used to control execution of the loop statement, referred to as the loop control variable, and *init*, *final* and *step* are expressions. Upon LOOP entry, the expressions *init, final* and *step* are calculated. If the second form of the statement is used (where *step* is omitted), *step* defaults to 1. *Var* is then assigned the value of *init* and the program proceeds normally until the ENDLOOP statement is reached. *Var* is then incremented by the value of *step* and checked against the value of *final.* If this value has been reached or exceeded, the program proceeds with the statement immediately following the ENDLOOP. Otherwise the control transfers to the statement following the LOOP statement, and the above process is repeated.

Nested LOOP structures are allowed. For example the following is an acceptable part of an MCL program.

```
LOOP A FROM 1 TO 10
  LOOP B FROM 4 TO 19 BY 0.1
    XT = XT * B + A
  END LOOP
END LOOP
```

Each LOOP statement must have a corresponding ENDLOOP statement, which can be written either as: ENDLOOP or as: END LOOP

If a variable is used as a loop control variable, it should not be used as a control variable in another loop nested inside the first one. Loop control variables should not be assigned any value, via an assignment or GET statement inside the loop they control. The only way to enter a LOOP is by its initial LOOP statement. Attempting to transfer the control inside the loop-block, from the outside, will result in an error.

Block-ifs and loops should be properly nested. This means that:

i) all block-ifs inside a loop-block must be terminated (by an ENDIF statement), before the corresponding ENDLOOP is encountered, and

ii) all loops inside a block-if must be terminated (by an ENDLOOP statement) before the corresponding ELSE or ENDIF is found.

B) The EXIT statement

This statement is used to exit a loop and transfer program control to the statement immediately following the corresponding ENDLOOP. Its syntax is simply:

```
EXIT
```

An example of its use follows:

```
LOOP I FROM 1 TO 10
  XT = XT * I
  WHEN I :=: A JUST EXIT
ENDLOOP
```

EXIT can be used only inside a LOOP-block.

2.2.6. MERLIN statements

These statements instruct MERLIN for the route of action and are in direct correspondence with the synonymous MERLIN commands an interactive user would issue to guide the minimization process.

There are parametric and non-parametric MERLIN statements. The EXECUTE statement, that has no synonymity with any of the MERLIN commands, is in close connection with the MERLIN macros and it is described at the end of this section.

a) Non-parametric MERLIN statements

These are: **ADJUST, FULLBACK, HALFBACK, NOBACK, ANAL, NUMER, FULLPRINT, HALFPRINT, NOPRINT, INDEX, NAME, MODEDIS, SHORTDIS, VALDIS, CATALOG, MEMO, LOOSALL, STEPALL, RESET, REVEAL, PANELON, PANELOFF, STOP** and **RETURN**.

The syntax format is:

CommandName

where *CommandName* is any of the above statements.

b) Parametric MERLIN statements

These statements use order independent parameters and are distributed among the following catagories:

(i) simple parametric statements,
(ii) special parametric statements and
(iii) special assignment statements.

Each parameter may be specified at most once.

(i) Simple parametric statements

These are the statements: **RANDOM, CONGRA, DFP, BFGS, ROLL, SIMPLEX, GRAPH, AUTO** (associated with panels) and
GRADCHECK, GRADDIS, ACCUM, PICK, REWIND, HIDEOUT, DISCARD, QUIT (associated with input parameters).

Their syntax format is:

$$CommandName(K_1 = V_1;\ K_2 = V_2;,\ldots,;\ K_N = V_N)$$

CommandName can be any simple parametric statement, K_1, $K_2,\ldots,$ K_N are keywords identifying either the panel or other input parameters, V_1, $V_2,\ldots,$ V_N are either expressions or filenames depending on the nature of the keyword.

For the panel-associated statements as well as for the GRADCHECK and GRADDIS statements, one needs not to specify all of the keywords since there exist default values. For the rest of them all keywords have to be specified.

In table 6, we list the identifiers for the panel-associated statements, in the same order as they appear in the MERLIN-Panels. Similarly in table 7, we list the identifiers for the statements ACCUM, PICK, GRADCHECK, GRADDIS, REWIND, HIDEOUT, DISCARD and QUIT along with their allowed values.

A few examples follow:

ACCUM(NOC = 1000;**NOP** = 3;**TARGET** = 1.0E–18)

DFP(NOC = 300;**TOL** = 0.06)

For statements associated with only one input parameter, the corresponding keyword may be omitted. For example the statements: **GRADCHECK(**0.00018) and **GRADCHECK(ERR** = 0.00018) are equivalent.

Table 6

Statement	Keys
RANDOM	**NOC,VEX,STEP,CSIZE,FAIL,LINE,PRI,CANCEL**
CONGRA	**NOC,TOL,ERR,USEG,PRI,WALL,CANCEL**
BFGS	**NOC,TOL,ERR,USEG,USEH,PRI,WALL,CANCEL**
DFP	**NOC,TOL,ERR,USEG,USEH,PRI,WALL,CANCEL**
ROLL	**NOC,TOL,STEP,FAIL,PRI,WALL,CANCEL**
SIMPLEX	**INIT,INITOL,INCALL,TOL,NOC,PRI,CANCEL**
GRAPH	**VAR,NOP,FROM,TO,LINES,COLS,CANCEL**
AUTO	**NOC,TARGET,CANCEL**

Table 7

Statement	Keys	Type
ACCUM	**NOC,NOP,TARGET**	Expression
PICK	**REC**	Expression
	FILE	File name
GRADCHECK	**ERR**	Expression
GRADDIS	**ERR**	Expression
REWIND	**FILE**	File name
HIDEOUT	**FILE**	File name
DISCARD	**FILE**	DISPO,STORE,BACKUP
QUIT	**FLAG**	Expression

As a further illustration, a portion of an MCL program involving the GRAPH statement follows.

```
LOOP I FROM 1 TO DIM
  AROUND = X[I]/1000.
  GRAPH(VAR = I;FROM = X[I] - AROUND;TO = X[I] + AROUND)
END LOOP
```

(ii) Special parametric statements

These are the statements: **DEMARGING, FIX, LOOSE** and **INIT**. They obey the following syntax:

$$CommandName(K_1 \cdot V_1;\ K_2 \cdot V_2; \ldots; K_N \cdot V_N)$$

The specifiers K_1, $K_2, \ldots, K_N$, can be:
X for the FIX and LOOSE statements,
L or **R** for the DEMARGIN statement, and
X or **L** or **R** for the INIT statement.
V_1, $V_2, \ldots, V_N$, are expressions for DEMARGIN, FIX and LOOSE, while for the INIT valid filenames.

Note that in the INIT statement each specifier can be used only once. For example the statement: **INIT**(**X**.FIL1;**X**.OTHER) is illegal.

A few examples follow:

```
FIX(X.1;X.5)            % Fixes the variables X[1] and X[5]
DEMARGIN (L.2;R.4)      % Removes the left margin for X[2] and the right margin for X[4]
LOOSE(X.7;X.1)          % Looses variables X[7] nand X[1]
INIT(X.STARTX;L.LMAR)   % The arrays X and L are assigned values residing in the files
                        % STARTX and LMAR correspondingly
```

(iii) Special assignment statements

These are the statements POINT, STEP, MARGIN and GODFATHER, and obey the following syntax:

$$CommandName(K_1.V_1 = E_{x1};\ K_2.V_2 = E_{x2}; \ldots;\ K_N.V_N = E_{xN})$$

K_1, $K_2, \ldots, K_N$, are:
X, for the POINT and GODFATHER statements,

L or **R**, for the MARGIN statement and,
S, for the STEP statement.
$V_1, V_2, \ldots, V_N$, are expressions.
$E_{x1}, E_{x2}, \ldots, E_{xN}$, are either expressions or, in the case of the GODFATHER statement, valid MERLIN names.

One may wonder why instead of using simple assignment statements, we employ the POINT, MARGIN and STEP statements to assign values to the corresponding intrinsic arrays **X**[1 : DIM], **L**[1 : DIM], **R**[1 : DIM] and **STEP**[1 : DIM]. The reason is that changing values of the intrinsic arrays **X**[1 : DIM], **L**[1 : DIM] and **R**[1 : DIM] is quite dangerous. MERLIN has several safeguards to protect the user from such traps. In the case of array **STEP**[1 : DIM], there is no danger, and it could have been done through a simple assignment statement. However, the philosophy we followed throughout was to drive MERLIN's operating system, without affecting its internal structure.

Some examples follow:

```
GODFATHER(X.1 = PRESSURE; X.2 = VOLUME; X.3 = TEMPERTUR)

POINT(X.1 = 7; X.2 = SQRT[4 * 2 + 1]/3)

MARGIN(L.1 = 8; R.I * * 2 = 19)

STEP(S.1 = 5; S.2 = 0.001)
```

c) The EXECUTE statement

This statement is in close connection with the MERLIN macros. Embeds a preconstructed macro in the code at compile time. The syntax is as:

EXECUTE *macroname*

or

EXECUTE *macroname* **FROM** *FileName*

where *macroname* is a MERLIN macro-name as described in ref. [1] and *FileName* is the name of the file where the macro resides.

In the first format, where there is no filename specification, MERLIN's file MACROF is assumed.

MCL programmers do not really need to use preconstructed macros, since they can develop easier and more efficiently any strategy that may be included in a macro. However, in extended versions of MERLIN, in which a user has added his own commands, the EXECUTE statement is the only way to embed these commands in an MCL program.

2.3. *An MCL program for automatic minimization*

The program shown in table 8, is the MCL implementation of MERLIN's automatic procedure invoked by the AUTO command. The strategy followed, whose flow chart is shown in fig. 1, is described below.

The methods BFGS, ROLL, SIMPLEX and RANDOM are invoked one after the other. For each of them, a rate is calculated by dividing the relative drop in the function's value by the number of calls spent. The method with the highest rate is then invoked again. The same procedure is applied repeatedly. If all rates assume vanishing values, then all the method tolerances are set to zero and the methods are applied in the following succession:

ROLL,RANDOM,BFGS,SIMPLEX.

Table 8

```
>   AN MCL ALTERNATIVE TO THE AUTO STRATEGY.
VAR TOTAL_CALLS; TARGET_VAL; CALLS; VAL_BEFORE; WORK_POSITION
VAR MEAN_RATE; RATE[1:4]; BEST_RATE; DENOMINATOR
DISPLAY '.. ENTER TOTAL CALLS ALLOWED AND TARGET VALUE'
GET TOTAL_CALLS; TARGET_VAL
VAL_BEFORE = VALUE
CALLS = T_COUNT
RESET
LOOP:
   BFGS(NOC = 300)
      WORK_POSITION = 1
      MOVE TO DETAILS
      RATE1_READY:
   ROLL(NOC = 300)
      WORK_POSITION = 2
      MOVE TO DETAILS
      RATE2_READY:
   SIMPLEX(NOC = 300)
      WORK_POSITION = 3
      MOVE TO DETAILS
      RATE3_READY:
   RANDOM(NOC = 300)
      WORK_POSITION = 4
      MOVE TO DETAILS
      RATE4_READY:
   BEST_RATE = MAX[RATE[1],RATE[2],RATE[3],RATE[4]] % GET HIGHEST RATE.
   WHEN BEST_RATE :=: RATE[1] JUST BFGS      % INVOKE AGAIN,
   WHEN BEST_RATE :=: RATE[2] JUST ROLL      % THE METHOD WHICH,
   WHEN BEST_RATE :=: RATE[3] JUST SIMPLEX   % SEEMS TO WORK IN
   WHEN BEST_RATE :=: RATE[4] JUST RANDOM    % THE BEST WAY.
   WHEN VALUE <=TARGET_VAL JUST FINISH
   MEAN_RATE = MEAN[RATE[1],RATE[2],RATE[3],RATE[4]]
   IF MEAN_RATE <=0.00005 THEN
      ROLL(NOC = 350;TOL = 0)
      RANDOM(NOC = 5000;FAIL = 20)
      BFGS(NOC = 350;TOL = 0)
      SIMPLEX(NOC = 350;TOL = 0)
   ELSE
      WHEN T_COUNT-CALLS < TOTAL_CALLS JUST MOVE TO LOOP
   ENDIF
FINISH % MCL PROGRAM USED ALL CALLS ALLOWED, OR RATES TOO LOW.
>
DETAILS: % THE RATE CALCULATIONS ARE PERFORMED HEREIN.
   WHEN VALUE <=TARGET_VAL JUST FINISH
   DENOMINATOR = ABS[VAL_BEFORE] * P_COUNT+PRECISION
   RATE[WORK_POSITION] = ABS[VALUE-VAL_BEFORE]/DENOMINATOR
   RESET
   VAL_BEFORE = VALUE
   WHEN WORK_POSITION:=:4 JUST MOVE TO RATE4_READY
   WHEN WORK_POSITION:=:3 JUST MOVE TO RATE3_READY
   WHEN WORK_POSITION:=:2 JUST MOVE TO RATE2_READY
   MOVE TO RATE1_READY
```

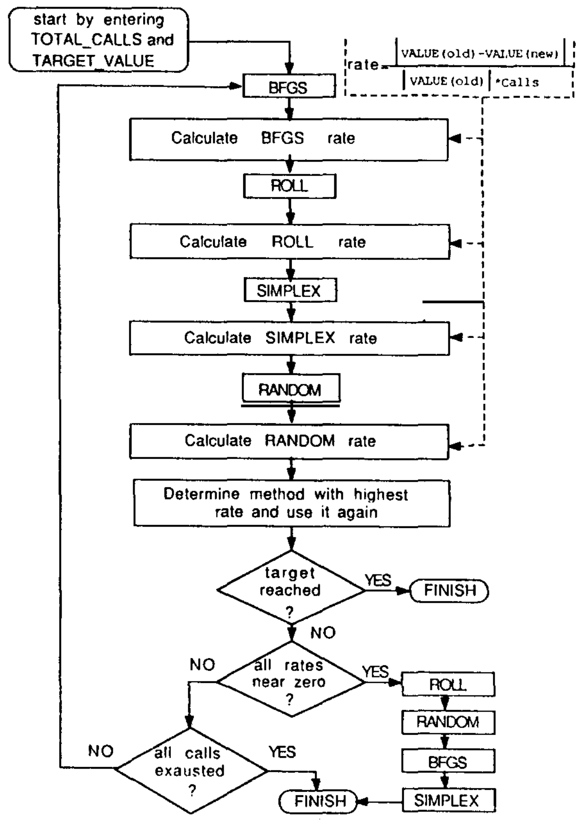

Fig. 1. Flow chart of the MCL implementation of MERLIN's automatic procedure.

One may observe that the MCL program is considerably more compact and easier to follow than the corresponding MERLIN's Fortran-code (subroutine MAGIC). This is a general rule. It is much easier to write MCL programs, than FORTRAN additions to MERLIN.

2.4. *A tactic in MCL handling constraints*

The following is a naive MCL program, for a three-variable problem with one equality constraint. We follow the suggestion given in Merlin's [1] manual. If the problem is:
Minimize $f(x, y, z)$, subject to: $g(x, y, z) = 0$,
we construct the function: $F(x, y, z, t) = f(x, y, z) + t * (g(x, y, z))^2$
Fix the t-parameter to a positive value and minimize F with respect to x, y, z. Then enhance t and repeat.

```
> SIMPLE-MINDED MCL PROGRAM FOR ONE EQUALITY CONSTRAINT
VAR I; SUM
POINT(X.1 = 3;X.2 = 2;X.3 = 55;X.4 = 0.1) % initialize variables
PANELOFF % disable panel prompting
EAT:
LOOP I FROM 1 TO 10
   POINT(X.4 = 10 * X[4]) % set the fourth parameter
   FIX(X.4) % fix the fourth parameter
   BFGS(NOC = 500) % invoke the BFGS method.
   SIMPLEX % invoke the SIMPLEX method.
ENDLOOP
> Calculate the sum of the absolute values of the gradient components to use it as a termination criterion.
SUM = 0.
LOOP I FROM 1 TO 3
   SUM = SUM + ABS[GRAD[I]]
ENDLOOP
>
IF SUM < 1.E-3 THEN
   SHORTDIS  % print results
   FINISH    % done.
ELSE
   STEPALL  % calculate search steps
   ROLL(NOC = 500)  % invoke the ROLL method
   DFP(NOC = 500)  % invoke the DFP method
   MOVE TO EAT  % repeat
ENDIF
```

3. Description of the MCL compiler

A detailed description of the MCL compiler package (MCLCOM 1.0) can be found in the developers manual (see appendix). Only some general features are discussed here.

The compiler is written in ANSI-FORTRAN 77 and it is truly portable. It can be considered as a two-pass compiler. During the first pass, four operations take place on each line of the MCL program, with the exception of comments and blank lines which are skipped. Packing, that eliminates irrelevant (to the compiler) blank spaces and underscores; lexical analysis, that transforms each line into tokens (syntactic entities that ease the code generation); parsing, that performs syntax-checks and arranges for appropriate error-messages, and assembling, that creates the basic object code, which however may not be executable at this stage (assembling and parsing are closely interrelated). The second pass, takes care of the loops, of the jumps and of the associated labels, checks the validity of the nesting structures and generates the MERLIN Object Code (MOC), which is executable from the new MERLIN-2.0 package [2].

Three files are associated with the compiler

(i) The file where the MCL-program resides.
(ii) The file where the MOC is disposed at.
(iii) The error list file.

The error list file, contains the incorrect lines (if any) of an MCL program, along with a brief explanation of the errors. When incorrect lines have been detected, MOC is not generated.

The file-names for these three files may be specified when MCLCOM is invoked, together with some additional parameters which define the compiling environment. The additional parameters are:

(i) BOUNDS

which if selected, the generated object code, performs special checks to prevent array boundary violation.

(ii) DEBUG

which if selected, each time a run-time error occurs, MERLIN 2.0, apart from the error message, will issue additional information concerning the line of the MCL source program, which caused the error.

Acknowledgements

The authors would like to thank Dr. G.A. Evangelakis, Dr. I.C. Demetriou, T.M. Liakopoulos and Prof. I.N. Demetropoulos for illuminating discussions.

Appendix. Description of the developers manual

This manual is a technical reference to both MCL and its compiler. The complete syntactic definition of MCL (in Extended Backus Naur Form) can be found there, together with a detailed description of the compiler, and with modification and installation directives. All routines are described one by one. An informal description of the MOC instruction set is given. Furthermore a complete list of the error messages is given together with their meaning and correction hints.

References

[1] G.A. Evangelakis, J.P. Rizos, I.E. Lagaris and I.N. Demetropoulos, Comput. Phys. Commun. 46 (1987) 401.
[2] D.G. Papageorgiou, C.S. Chassapis and I.E. Lagaris, Comput. Phys. Commun. 52 (1989) 241.