Simulation of Systems, L. Dekker, editor

MACRO-HANDLER FOR SIMULATION PACKAGES USING ML/I

François E. Cellier

Institute for Automatic Control
The Swiss Federal Institute of Technology
Physikstr. 3
CH-8006 Zurich ; Switzerland

The demands on a macro-handler for use in simulation languages are described. It is shown how general purpose macro generators may be used in this context. An introductory part of the paper shows under which circumstances such macro-handlers are useful. The possibilities for deviding simulation packages into several subpackages are discussed, out of which the macro-handler will be one. Prospectives for further development of simulation languages suitable for use in parameter identification and other optimization problems, interactive computer aided design problems etc. are pointed out. Since such programs for most applications will have to run on minicomputers an independently working macro-handler as described below will be one element of it.

1. INTRODUCTION

In 1967 the SCi Simulation Software Committee published the CSSL-Report [1]. This report described a new simulation language far beyond any language propagated before but at that time existing only as a goal for further development since no compiler existed to run CSSL on any installation. Since this report has been widely accepted newer packages base very much upon it. The newest representative of this "family" of simulation packages is ACSL [2] being even a superset of CSSL. The enormous comfort of such simulation packages has to be paid for by high requirements of core memory and a rather slow execution compared to the execution of a equivalent program which is e.g. entirely coded in FORTRAN-IV. Besides, the compilers to these packages - if developed on a commercial basis - will cost over $ 10000 which makes it more and more attractive to the occasional user of such a package to pay high amounts for data transmission (via satellite) to be connected to a computer placed anywhere on our globe rather than to buy the compiler for his own installation.

Although there exists a great demand for these packages they are not really suited if:

a) discontinuous functions form part of the model as explained in [3]

b) interactive use is required, since all of these packages run on "big" computer installations. Interactive utilization of user programs (written in high level languages) on such installations is uneconomic for most applications

c) a simulation study exceeds a - very! - limited number of single simulation runs, since the costs for one simulation run will always be so high that e.g. parameter identification problems for several unknown parameters requiring hundreds of simulation runs would normally not pay out. Also for this application the simulation should, therefore, be carried out on a minicomputer.

The problem can be described as to find a way to split up the simulation package into independently working subunits (modules) to allow a maximum number of features to be coded within e.g. 28k of core memory on a 16 bit machine.

The "classical" CSSL-type simulation package consists of two programs:

a) a preprocessor which interprets the user supplied parallel code and translates it into procedural code written in an intermediate language (e.g. FORTRAN-IV)

b) a run-time program which reads in the actual simulation parameters, carries out the simulation, interprets output request statements and produces output accordingly.

These two programs always work independently. If smaller modules are required it is possible to cut down both programs in the following way: Program (a) may be subdivided into:

α) a macro-handler which can be coded independently of the rest since anyhow all macro calls have to be replaced by their definition bodies before any further preprocessing is activated to guarantee for modularity. This is shown in [4]

β) one or several preprocessors for the different sections of the user's program where section means a functional program block (e.g. derivative section). It may be useful to have different preprocessors for different sections, since each section may define a "language" of its own with a syntax of its own according to its specific duty

γ) a preprocessor for introduction of header information which is necessary to allow several sections to have automatically updated common variable lists. A statement

COMMON *B* (1)

placed at the beginning of section A should cause the header information (≙ specification statements) generated by the preprocessor of section B to be placed at the disposal of section A.

Program (b) may be subdivided into:

α) a program for reading in the actual parameters and for carring out the simulation (data input may be overlayed if necessary)

β) a program for processing of output requests (postprocessor).

The most successful steps towards the concept described above so far published have been made at the University of Arizona where the DARE simulation language family has been developed. The packages out of this family are described in [5,6]. A batch version DARE-P [7] has been implemented by the author of this paper for use on a PDP-11/45 running under DOS/BATCH versions 9 and 10 within 28k of core memory without need for overlay structure. This could be achieved since DARE-P is structured as above consisting of three independent programs for preprocessing, run-time and postprocessing. DARE-P has no macro facility available, but beside this shortcoming is almost as powerful as CSMP-III and has even certain advantages over CSMP-III (as for example the "logic"-section being more flexible than the corresponding "initial"- and "terminal"-sections of CSMP-III). The macro-handler described below has been used to ameliorate the possibilities of DARE-P, but is not restricted to this package. Since it is completely modular it may be used in connection with any other package as well.

Up to now only standard features of simulation packages have been considered. The value of such a package could, however, remarkably be augmented if:

a) a nonlinear programing package would be overlaid to the run-time program (b_α) for automatic minimization of a performance index which is to be specified in a seperate section of the user's program.

b) an additional program would be added at the end of the program chain for interactive input of new simulation data and/or output request statements.

Taken all together we obtain the following program structure:

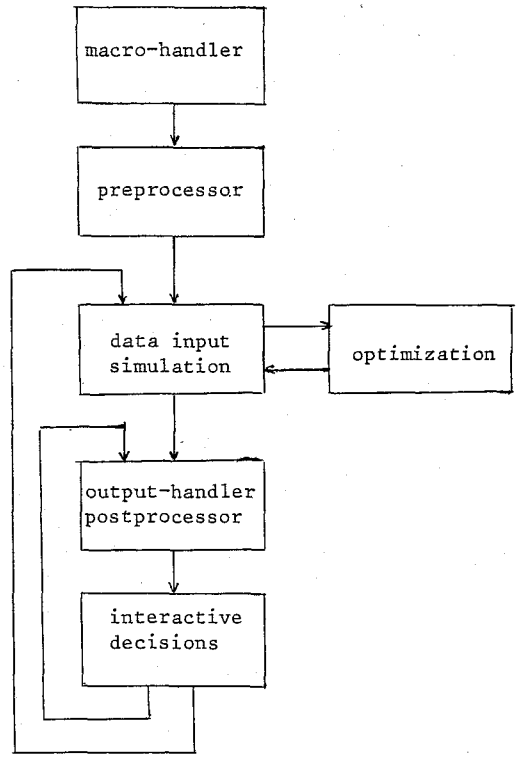

fig.1 program structure of a simulation package to run on a minicomputer

By coding the programs for this package the following should be taken into account:

a) the intermediate program should be generated in a well readable form such that skilled users have the possibility to code their programs directly in the intermediate language itself bypassing all preprocessing activities. This is realized e.g. in DARE-P but not in CSMP-S/360.

b) the generated run-time program could as well be of the GASP-type [3] to allow a broader range of applications. The different subprograms of the intermediate program would be generated from different sections of the user's program.

c) their could be several different preprocessors or maybe several layers of preprocessors made available from which the user may select those best fitting his problem (e.g. preprocessors for analysis of electrical networks, pert networks etc.).

There is a lot of work still to be done until this goal is achieved. To start with, this paper

describes the first program block which is the macro-handler. It is shown that a general purpose macro generator (the author selected ML/I) can be utilized and what special problems have to be considered.

2. SELECTION OF COMPILER LANGUAGE

According to the author's opinion it is extremely important to select an adequate language for the coding of the macro-handler, more than for any other of the programs denoted above. The macro-handler of CSMP-III, for example, which is coded in FORTRAN-IV may only be used for coding of dynamic functional blocks (which is, of course, the most important part in this context). However, execution is rather slow.
A more universal *interpretative* macro-handler as offered in CSSL-III executes extremely slow and needs very high amounts of core memory.
A macro-handler coded directly in macro assembly or in a general purpose macro language which itself is coded in assembly will execute between 10 and 100 times faster while occupying much less core memory. Unfortunately, there does not exist any "universally accepted" macro language. The macro-handler will thus be installation dependent. This disadvantage should, however, be more than equalized by the advantages described above. Besides, there is no other way if a powerful macro-handler should be coded for use on a minicomputer with restricted amount of core available.

For the PDP-11/45 computers there exist two such packages: ML/I [8,9] and STAGE-2 [8] out of which ML/I has been chosen. By use of STAGE-2 it would have been difficult to spread a macro call over several lines, since STAGE-2 is line-oriented. This feature is important in simulation techniques applications since often the number of arguments of such macros will be relatively high.

3. THE MACRO ENVIRONMENT

As a first step one has to come to an agreement concerning the macro environment that is grammar and syntax of the macro calls and their definition bodies.

Each macro call shall have the following form:

$$\text{name } (y_1,y_2,\ldots,y_k=x_1,x_2,\ldots,x_m) \qquad (2)$$

with name being the name of the macro, y_i being formal output parameters and x_i being formal input parameters. The distinction between input and output parameters has no functional meaning. It ameliorates the readability of the programs and thus serves for the users convenience. Input and output parameters are called *arguments* of the macro.

Parameter lists may be continued over several lines by simply breaking the statement after any "," or "=":

$$\text{name } (y_1,y_2,\ldots,y_k=x_1,x_2,\ldots,x_i, \quad x_{i+1},\ldots,x_m) \qquad (3)$$

In ML/I this can be coded by giving each macro definition an appropriate *delimiter structure*, describing how the arguments of the macro are separated from each other. Since the delimiter structure of all macros should be the same (as described above) the delimiter structure will be coded as a macro itself. The appropriate coding in ML/I will be:

```
MCDEF <PARENS>
AS <WITHS ( N1 OPT , N1 OR , WITH NL N1 OR ,
WITH NL WITH SL N1 OR = OR = WITH NL OR = WITH
NL WITH SL ALL N2 OPT , N2 OR , WITH NL N2 OR ,
WITH NL WITH SL N2 OR ) WITH NL OR ) WITH NL
WITH SL ALL>
```

WITHS (belongs to the macro name (since the first argument y_1 starts only behind the opening bracket). N1 denotes a label. The words OPT and ALL embrace a list of several possible delimiters separated from each other by the word OR which are:

```
,
, WITH NL    (, followed by a new line)
, WITH NL WITH SL (, followed by a
          new line and a start line)
=
= WITH NL
= WITH NL WITH SL
```

The combination of "," or "=" with the new line as one delimiter allows the continuation of argument lists over several lines. After each delimiter starting with a "," there is again a label N1 showing back to the beginning of the option list. The first N1 is therefore a label whereas later uses of N1 (in this example) denote a "go to"-statement. The distinction between those two possible uses is done in accordance with its position in the string. This feature of ML/I allows definition of macros with a variable number of arguments (as required for this purpose).

Any further macro definition can now be written as:

```
MCDEF <name> PARENS
AS <....>
```
(4)

Macro calls are only to be detected as such if they are not placed within a comment line. For use in DARE-P they are lines marked by a C or by an * in column 1. This can be coded by a command to skip over each line starting with one of those symbols:

```
MCSKIP DT, OPT SL WITH C OR SL WITH * ALL NL
```

All text is skipped over which is included in the two delimiters "start-line followed by a C" or "start-line followed by an *" and "new line". The two parameters D and T cause both the delimiters and the text between to be copied over to the output file.

In the following macro call:

```
NAME1 (A=AMAX1(B,AMIN1(C,D)),E)          (5)
```

the whole expression AMAX1(B,AMIN1(C,D)) logically describes one argument. Syntactically the following "arguments" would be found:

- A
- AMAX1(B
- AMIN1(C
- D

and the string "),E)" would not belong to the macro call. To avoid this missinterpretation the following statement may be used:

```
MCSKIP DTM, ()
```

Text between parenthesis will be skipped over with text and delimiters copied to the output file. The search for the delimiters is done on a matched basis. The opening parenthesis of the macro call will not be detected since it belongs to the macro name (ML/I always searches for the longest possible text strings).

To delay the insertion of macros until a later time the statement

```
MCSKIP TM, <>
```

is used, which causes a skip over a text string encompassed by the two matched delimiters "<" and ">". The text is copied to the output file whereas the delimiters are deleted. Text in a macro definition should always be placed between those brackets to cause inserts to be made only at *macro replacement time* and not at *macro definition time*.

Inserts are characterized by the encompassing symbols "%" and "." using the statement:

```
MCINS %.
```

The statement:

```
MCSKIPG £
```

allows the user to separate two text strings consisting of only alphanumeric information from each other by placing the symbol "£" between which is deleted immediately.

The complete macro environment body has now the following form:

```
MCSKIP TM,<>
MCSKIP DTM,()
MCSKIPG £
MCSKIP DT, OPT SL WITH C OR SL WITH * ALL NL
MCINS %.
MCSET P1=99
MCSET P2=9999
MCSET P3=0
MCSET P4=0
MCDEF <PARENS>
AS <WITHS ( N1 OPT , N1 OR , WITH NL N1 OR , WITH
NL WITH SL N1 OR = OR = WITH NL OR = WITH NL WITH
SL ALL N2 OPT , N2 OR , WITH NL N2 OR , WITH NL
WITH SL N2 OR ) WITH NL OR ) WITH NL WITH SL ALL>
```

The meaning of the MCSET-statements will be explained in the next chapter.

4. SPECIAL PROBLEMS OF SIMULATION MACROS

The output file of the macro-handler still consists of *parallel information* that is a set of statements which have not yet been rearranged into their appropriate order at execution. This implies that all variables may only be defined once (may only appear once at the left side of the equal sign). To allow macros to be inserted several times, all locally defined variables within the macro definition body (≙ all variables appearing on the left side of the equal sign and not belonging to the formal parameter list) have to obtain new, unique names each time they are evaluated. This can be achieved by giving each such variable a name consisting of the two symbols ZZ followed by a unique number. The statement:

```
MCSET P1=99
```

in the macro environment body assigns the value 99 to the system variable P1. The i-th locally defined variable of a macro definition body may now be represented as ZZ%P1+*i*.. At macro replacement time the two letters ZZ followed by the inserted actual value of the expression P1+*i* will be copied over to the output file. The first locally defined variable will, therefore, obtain the name ZZ100. At the end of the macro definition body the system variable P1 has to be augmented by the number of locally defined variables k of the macro definition. For this purpose the following statement may be used:

```
MCSET P1 = P1 + k
```

The same procedure can be used for the generation of local labels in a procedural text string. Also these labels have to be unique since all of them will after translation belong to the same subroutine (DIFEQ1 in the case of DARE-P). For

this purpose the system variable P2 is used generating labels from 1000 upwards. %P2+*i*. will represent the i-th local label of the macro definition.

As an example let us consider the model of a DC-motor described by the following state equations:

$$\dot{\underline{x}} = \begin{bmatrix} -R_a/L_a & -c_m/L_a & 0 \\ c_m/J_m & -c_f/J_m & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \underline{x} + \begin{bmatrix} 1/L_a \\ 0 \\ 0 \end{bmatrix} \cdot u$$

where: x_1 = current through the motor
x_2 = motor speed
x_3 = motor angle
u = voltage over the motor
R_a = motor resistance
L_a = motor inductivity
c_m = back-EMF constant
c_f = friction constant
J_m = motor inertia

By using the macro-handler of CSMP-III this model would be coded in the following way:

```
MACRO  X3, X2, TM = DCMOT (UIN)
X1 = INTGRL (0.0, X1DOT)
X2 = INTGRL (0.0, X2DOT)
X3 = INTGRL (0.0, X3DOT)
X1DOT = (1./LA)*(UIN - X1*RA - CM*X2)
X2DOT = (1./JM)*(TM - CF*X2)
X3DOT = X2
TM = CM*X1
ENDMACRO
```

Three kinds of variables can be distinguished: formal parameters, locally defined variables and global constants. Table 1 describes them:

number	formal parameter	local def. var.	global const.
1	X3	X1	LA (ALA for use in DARE-P)
2	X2		RA
3	TM		CM
4	UIN		CF
5			JM (AJM for use in DARE-P)

table 1: Variables of the macro describing a DC-motor.

By use of ML/I the following macro definition body (for a DARE-P program) can be used:

```
MCDEF <DCMOT> PARENS
AS<MCNOSKIP£ZZ%P1+1..=(1./ALA)*(%A4.-ZZ%P1+1.*RA
                                  L  -CM*%A2.)
      %A2..=(1./AJM)*(%A3.-CF*%A2.)
      %A1..=%A2.
      %A3.=CM*ZZ%P1+1.
MCSET P1 = P1 + 1
>
```

A macro call:

```
      DCMOT (XX3,XX2,TTM=UUIN)                    (6)
```

will result in:

```
      ZZ100.=(1./ALA)*(UUIN-ZZ100*RA-CM*XX2)
      XX2.=(1./AJM)*(TTM-CF*XX2)
      XX3.=XX2
      TTM=CM*ZZ100
```

The first three statements of the CSMP-macro are not necessary here since the equation:

$$\dot{x} = a \cdot x \qquad (7)$$

which is coded in CSMP-III as:

```
X = INTGRL (0.0, XDOT)
XDOT = A*X
```

is represented in DARE-P by:

```
      X. = A*X
```

The above example illustrates furthermore the coding of formal parameters (≙ attributes) of the macros. TM which is the third attribute of the macro is coded as: %A3.

The following example shows proper treatment of memory- and history-functions (≙ functions requiring a certain number of storage allocations for backup memory). The statement:

```
      HSTRSS (Y = YIC, P1, P2, X)
```

should result in:

```
      Y = HSTRSS (k, YIC, P1, P2, X)
```

where k is a pointer pointing to the first memory cell of a stack being used for this specific call of the history-function HSTRSS. For this purpose the system variable P3 is used which is augmented at the end of the macro definition body by the number of storage allocations required by the function (3 in the case of the HSTRSS-function).

The macro for the HSTRSS-function may be coded

as:

```
MCDEF <HSTRSS> PARENS
AS<MCNOSKIP£%A1.=%WDO.%P3+1.,%A2.,%A3.,%A4.,%A5.)
MCSET P3 = P3 + 3
>
```

In this example it is also shown how to use the macro name in the macro definition body itself. %WDO. is the written (≙ not evaluated) delimiter number "0" which is the delimiter in front of the first argument → corresponding to the name of the macro followed by a left parenthesis.

5. ADDITIONAL USEFUL FEATURES OF ML/I

Up to now those features of ML/I have been described which are necessary in the context of macro facilities for simulation languages. In the following example there will be described a more complicated feature showing the *interpretative* use of ML/I.

In DARE-P all output requests are processed in the output-program which is executed only after the whole integration procedure is terminated. If -- during the simulation -- the program could not proceed any longer for an unknown reason it is sometimes difficult to find out how far the simulation could be carried out, since no output at all will appear. For this reason a subroutine PRINT should be coded which prints the contents of a stack onto the output device at each k-th evaluation of the derivative section. Another subroutine ARRAY should be used to fill a variable number of simulation variables into the stack . This is necessary since no indexed variables may be used in DARE-P. For this purpose the statement:

```
ARRAY (DUMMY = T, DT, X3, X2, TM, UIN)
```

should result in:

```
PROCED  ZZ137 = ZZ138
CALL ARRAY (T,1)
CALL ARRAY (DT,2)
CALL ARRAY (X3,3)
CALL ARRAY (X2,4)
CALL ARRAY (TM,5)
CALL ARRAY (UIN,6)
ENDPRO
```

This can be achieved by the following macro definition:

```
MCDEF <ARRAY> PARENS
AS<MCNOSKIP£PROCED  ZZ%P1+1. = ZZ%P1+2.
MCSET P5 = 1
%L1.      CALL %WDO.%AP5+1.,%P4+P5.)
MCGO L2 IF %T1.GR%P5+1.
      ENDPRO
MCSET P4 = P4 + P5
MCSET P1 = P1 + 2
MCGO L0
%L2.MCSET P5 = P5 + 1
MCGO L1
>
```

This macro is maybe more difficult to understand for the novice user of ML/I. %Li. is the i-th label. Its replacement text is none, but it may be used to mark a point of the macro to which during evaluation of the macro may be jumped. %AP5+1. is the argument with the number P5+1 (≙ actual value of the system variable P5 plus one). %T1. is the number of arguments of the specific call. MCGO L2 IF %T1.GR%P5+1. means, therefore; "continue at label L2 if the number of arguments of the macro is greater than the actual value of the system variable P5 plus one". MCGO L0 is a statement which corresponds to a RETURN-statement in a FORTRAN subprogram. The system variable P5 counts locally the arguments, P4 counts them globally. This allows to fill new variables into the stack by additional calls to the macro ARRAY.

6. RESULTS

By use of the technique described above there have been modeled machine tools in ML/I and DARE. The models of such machine tools normally consisted of more than 150 statements in more than 100 variables. The program runs on a PDP 11/45 under DOS/BATCH versions 9 or 10 within 28k of core memory without need for using overlay structure. Over 20 macros could be generated within 20 seconds of time whereas earlier programs coded in CSMP-III needed over 40 seconds on an IBM 370 installation for the same purpose. Interpretative macros are generated with almost the same speed as others. Equivalent macros coded in CSSL-III and in ML/I both computed on a CDC 6500 installation turned out to need 10 ÷ 100 times more execution time in the case of CSSL-III.

The author of this paper knows from his own experience that the coding procedure by using ML/I is problematic and that there exists a much higher probability for errors than by using CSMP-III or CSSL-III. Beside that the machine dependency is a great disadvantage of using ML/I. However, ML/I -- as other similar packages -- is very useful for the application of simulation techniques if:

a) the utilized language has no macro facility available (as in the case of DARE-P, MIMIC and others)

b) the macro generation by use of the own macro-handler turnes out to be to expensive and

c) the program has to work on a minicomputer on which the own macro-handler of the simulation language cannot be used, because of too high

reguirements for core memory.

ACKNOWLEDGMENT

The author wishes to express his gratitude towards AGIE Ltd. Losone/TI Switzerland for their scientific and financial support of the research described in this paper. Furthermore he wants to thank Prof.Dr.M.Mansour, the head of the Institute for Automatic Control at the Swiss Federal Institute of Technology Zurich, who gave him the opportunity to carry out this work.

REFERENCES

[1] *The SCi Continuous System Simulation Language (CSSL)* Simulation Vol.9 No.6 December 1967

[2] *Advanced Continuous Simulation Language.- User/Guide Reference Manual*. Mitchell and Gauthier Assoc., 1337 Old Marlboro Road, Concord Mass. 01742 USA.

[3] F.E.Cellier, Blitz A.E.: *GASP-V: A Universal Simulation Package*. Proceedings: AICA 1976, Delft, The Netherlands.

[4] F.E.Cellier, Ferroni B.A.: *Modular, Digital Simulation of Electro/Hydraulic Drives using CSMP*. Proceedings: 1974 Summer Computer Simulation Conference (SCSC), Houston Tex. USA.

[5] G.A.Korn: *Project DARE Differential-Analyzer Replacement by On-Line Digital Simulation*. Proceedings: AFIPS/FJCC 1969, AFIPS Press, Montvale N.J. 1969

[6] G.A.Korn: *New Techniques for Continuous-System Simulation*. Automatic Control Theory and Applications vol.2 No.1, Acta Press, Calgary, Canada, January 1974

[7] J.J.Lucas, Wait J.V.: *DARE-P User's Manual*. CSRL Report 255, University of Arizona, College of Engineering, Dept. of Electrical Engineering, Computer Science Research Laboratory, Tucson AR., USA 1974.

[8] P.J.Brown: *Macro Processors and Techniques for Portable Software*. John Wiley 1975.

[9] P.J.Brown: *ML/I Macro Processor*. DECUS Program Library. Decus No. 11-69. July 1972.