EVALUATION OF ACSL

by

M.T. Rahbar, M. Ekbatani, S. Bennett and D. A. Linkens

Department of Control Engineering

University of Sheffield

Mappin Street, Sheffield S1 3JD

Research Report No. 304

September 1986

Report prepared for British Gas Engineering Research Station, Newcastle-Upon-Tyne.

SUMMARY

This report evaluates the Advanced Continuous Simulation Language (ACSL). It describes the language structure and assesses its features.

The user-computer interaction, ease of use of the package and its capabilities in result analysis are discussed. The report also considers the ways to improve the software.

CONTENTS

1. INTRODUCTION

The Advanced Continuous Simulation Language (ACSL) is an equation oriented simulation language which is available for several different computers. This report is concerned with the IBM XT version which is being run on an Olivetti M24 SP computer in the Department of Control Engineering, University of Sheffield.

During a six month period various models (linear, non-linear etc.) have been simulated on ACSL by student users with widly divergent programming experience. This report assesses the package based on the experience gained in using the ACSL. It examines its ability to provide the features required by users in the Control Engineering Department and users at the British Gas Research Centre for their simulation studies. The requirements can be summarized as follows:

1) Ease of use; the language should cater for both novice and expert users.

2) Provision of extensive simulation functions; e.g. wide range of integration algorithms.

3) Ability to split a program into submodels and to create and use submodel libraries.

4) Extensive facilities to manipulate the data output from a simulation run.

A more general discussion is also made on the ways that the package may be improved. This, however, mostly concentrates on the user-computer interation, as this requirement is not satisfied by most (if any) of the existing simulation languages. The availablity of microcomputer to a wide spectrum of users has clearly put more emphasis on this requirement, especially in using technical programs such as simulation.

It should be noted that this report is concerned with evaluation of the PC version of the ACSL. Versions of ACSL for mini and mainframe computers offer a wider range of features e.g. more options for graphical display of results.

2. THE ACSL LANGUAGE

The basic structure of ACSL follows the specification established by the SCi Technical Committee on Continuous System

Simulation Language (CSSL) [1].

There are three REGIONS in the ACSL structure each of which may be further sub-divided into smaller conceptual units.

2.1 The Initial Region

The initial region is used for the execution of all those operations which must be performed before the simulation begins. In practice, the initial region is used for:

1) Setting of constant values for both model and experiment.

2) Setting of integration parameters.

3) Initialising output, e.g. printing headings.

4) (Re)initialising of system (state) variables to initial conditions.

For multiple simulation runs, the initial region is re-entered after each individual run for re-initialisation of those state variables which require it. This region contains procedural code (i.e. FORTRAN code) [2].

2.2 The Dynamic Region

The dynamic region is the active portion of ACSL simulation program, the heart of which is the Derivative Section. The derivative section contains the model equations which are evaluated under the control of the integration system. This section is strictly non-procedural.

The procedural parts of the dynamic region are used for those operations which do not have to be performed 'continuously' at each integration step. Typically, these periodic operations are:

1) Output of system variables.

2) Calculations which are dependent on the independent variable but which do not form an integral part of the equation set.

3) Testing of conditions to determine whether or not to

terminate the simulation.

The parameter which represents the period over which the integration system exercises control of the derivative section is the COMMUNICATION INTERVAL.

The statements in the derivative section need not be ordered but will be automatically sorted into the correct sequence so that intermediate values are calculated prior to their use.

2.3 The Terminal Region

When control is passed to the terminal region from the dynamic region, it implies that the simulation run has ended. Any final computations, which might be required, are performed and program execution is halted.

2.4 ACSL Structure Statements

The ACSL program skeleton is shown in Figure 1.

```
INITIAL
.................
Initial Region
.................
END

DYNAMIC
.................
Procedural part of Dynamic Region
.................
  Derivative
.................
  Derivative Section
.................
  END

END

TERMINAL
.................
Terminal Region
.................
END
```

Figure 1. ACSL Program Skeleton.

Note, if no structure statements are present (e.g. INITIAL), the program is to be treated as a single derivative section. If a region is not required, it can be omitted.

3. PROGRAM GENERATION AND EXECUTION

ACSL is implemented in FORTRAN. It acts more like a pre-processor as ACSL programs are first translated to FORTRAN which are then compiled by the FORTRAN compiler. The sequence of the program generation and execution is shown in Figure 2.

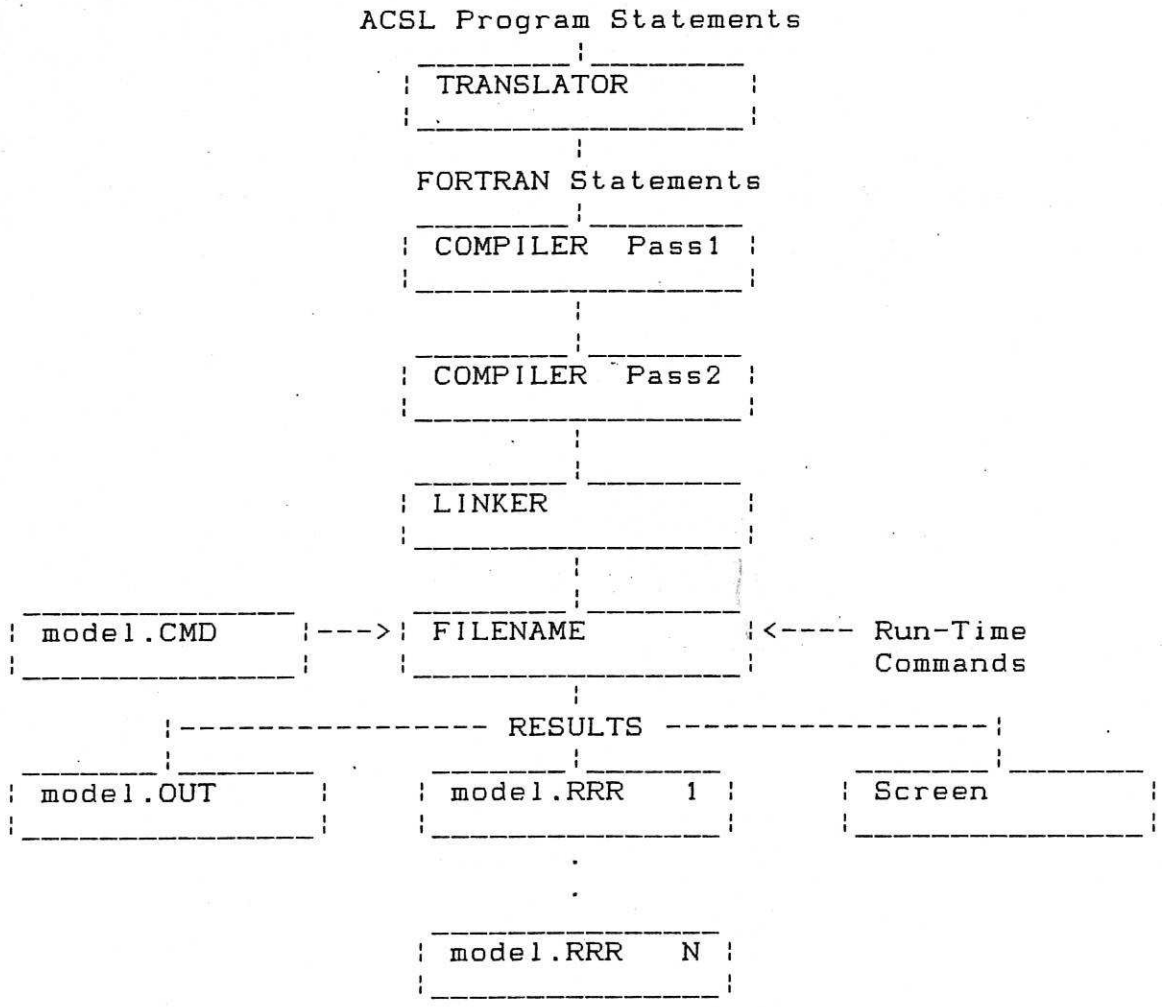

Figure 2. ACSL Program Generation and Execution.

A model.RRR file recieves binary data produced at run time. This data is used later to plot or print the results.

3.1 A Benchmark Test

The Pilot Ejection study was used to determine the speed of ACSL translator, compiler, linker and the execution run. The results are summarized in Table 1. A listing of the program is given in Appendix A.

Compilation	Time (SEC)
Translator	12.4
Comp. P1	15
Comp. P2	21
Linker	61
Run-Time	**Time (SEC)**
Tabulated O/P	110
Time Res. Plot	12.8
X, Y Plot	6

Table 1. Benchmark Test Results.

4. ACSL FEATURES

An ACSL simulation consists of two sections: a model definition and run time commands (or experimental definition). Using this structure, a continuous system is mathematically modelled with ACSL statements in the model definition section, then the model is analysed under the control of instructions interpreted in the run-time command section. The advantage of this structure is that once the model is defined, it can be saved on file and analysed indefinitely with any run-time commands specified interactively and/or in batch mode.

The basic facilities provided by ACSL for each section will be assessed in the discussion which follows.

4.1 Model Definition

4.1.1 Integration Algorithms

There are seven integration algorithms available in ACSL, as listed in Table 2. The integration algorithm can be changed by the SET command (section 4.2) at run time.

IALG	ALGORITHM
0	Sampled Data System
1	Adam's Moulton; variable step
2	Gear Stiff; variable step
3	Runge-Kutta First Order or Euler
4	Runge-Kutta Second Order
5	Runge-Kutta Forth Order
7	User Supplied Subroutine

Table 2. Integration Algorithms in ACSL.

The Adams-Moulton and Gear's stiff are both variable step, variable order, integration routines that are self-initializing. In general they will keep the per step error in each state variable below the desired value. Gear's Stiff integration method can take calculation steps that are orders of magnitude larger than the smallest time constant. There is an overhead involved, however, since a linearized state transition matrix must be formed and inverted. For models in which the range of time constants extends over one to two decades, there is little benefit in using this method: the Adams-Moulton technique is invariably faster. If the range of time constants covers more than three to four decades, then the Gear technique should be significantly faster than any other [3].

Integration algorithm zero is used to model discrete controllers with samplers controlling transfer to and from the continuous section.

More than one Derivative section may be used, each with its own independent integration algorithm and integration step size. Although this technique can save execution time when correctly used, any implementation must be approached with caution since, in general, incorrect answers will be obtained unless the model is split with full understanding of the effects of computation delays for variables that cross block boundaries. Novice users should use no more than one derivative section in their program unless it is to model a discrete controller where natural delays occur.

ACSL also provides functions for detection of discontinuities. Essentially, two features are required, (i) the discontinuity must be able to be specified as a discontinuity (or specified value) function and (ii) the integration algorithm must be able to adjust its step size under the control of some detection scheme to coincide with the occurance of the discontinuity. These features are supported by the functions DBLINT (double limited integrator) and LIMINT in ACSL.

Note, Partial Differential Equations (PDEs) can also be solved using Vector Integration.

4.1.2 Macro

The macro facility is the most powerful and advanced feature of ACSL. It allows the user to expand the language capabilities by defining new operators as the need arises.

A macro may be used in one of two essentially distinct ways. The first is akin to a subroutine or a function which is defined once and called from many places. The only way to define operators involving integrators is by using this Macro operator. The second approach is to define blocks and write all the equations in terms of standard nomenclature. Those blocks can become part of a system library (e.g. an Actuator).

Some of the weaknesses of the macro facility in ACSL are:

1) The macro argument list is long and complicated. For example:

```
MACRO VALVE ( qv1, pup, pint, temp, z, cg1 ...
              n7, cgmax, g, fk, xt, ymax, fp )

MACRO STANDVAL n7= ....,     cgmax= ....,
               g= ....,      fk   = ....,
               ..........
```

Also, any local constants used have to be declared in the macro heading.

2) Macros written to handle arrays are extremly hard to read since no mnemonic symbols can be used for the arguments [3].

3) Macro generates large numbers of dummy variables, for local variables at each invocation, which have no physical significance. An alternative approach is to use concatenation feature to build unique symbols. This is, however, confusing as symbols can be mixed. Using the concatenation feature the argument list is simple, but there is no flexibility, and arguments can not be expressions.

4) Care must be taken when a macro is called where expression are used, wrong answers can be obtained if parentheses are not placed around the argument. For example,

```
MACRO ACCUM ( tot, w1, w2, ic )
   tot= INTEG ( w1 - w2, ic )
MACRO END
```

at invocation time

```
ACCUM ( mass = win, wp1+ wp2, massic )
```

which would give the line of code

```
mass= INTEG ( win -wp1+ wp2 , massic )
```

which is wrong, since the wp2 has a plus sign in front of it. The macro above should have been defined by

```
MACRO ACCUM ( tot, w1, w2, ic )
   tot= INTEG ( w1 -( w2 ), ic )
MACRO END
```

to obtain

```
mass= INTEG ( win - ( wp1+ wp2 ), massic )
```

As the above discussion illustrates, the macro facility should only be used by experienced users.

4.1.3 Subroutine

ACSL programs may contain FORTRAN statements. The FORTRAN subroutines and functions are placed at the end of an ACSL program. They may be called from the program body, Macro or other subroutines. The use of subroutine has the following drawbacks:

1) The user has to learn FORTRAN and obey the FORTRAN programming rules (eg. statements start at column seven).

2) The ACSL translator does not check any of the FORTRAN programming codes. It, therefore, takes longer to find any errors in the program.

3) The user has to be familiar with the FORTRAN 'LIB' utility program to be able to create a library of subroutines where the subroutines are held in compiled form. He, then, can link an

ACSL program to the appropriate library.

4.1.4 Function Generators

One, two and three dimensional tables may be used to describe an arbitrary function. The interpolation between points is linear.

4.1.5 Algebraic Loop Solver

ACSL uses the Newton-Raphson iteration method to find a solution of simultaneous or algebraic equations. This operation, however, increases the run time considerably, as an iteration has to take place during each function evaluation.

4.2 ACSL Run-Time Commands

The commands, at run time, may be submitted either from a file (batch run) or interactively. In the latter case the user types the desired commands one by one, whereas in batch mode he only specifies the file which contains the appropriate run-time commands.

Commands can also be grouped as a PROCED routine, with nesting capabilities, and are invoked by its name. For example:

```
PROCED GOPLOT
START
PLOT x, y, z
PLOT v1, v2, v3
.
.
END
```

If one wants to repeat a simulation with different values for a constant, K (say) then command sequence for the set of runs changing K is now:

```
SET K= 1.0 $ GOPLOT
SET K= 2.0 $ GOPLOT
.
.
```

with significant saving of input volume.

Data can be set into any known constant array or variable by the SET command. This command would normally be used for

changing the values of constants. They stay that way until changed again. For example:

```
SET MERROR v1= 0.0001     ( Relative
SET XERROR v2= 0.0001       & absolute error bounds )
```

4.2.1 Output

In ACSL the output data from a simulation run can be tabulated and/or plotted. The graphical output is produced off-line. The data is first written to a file as the run proceeds. The user can then display these variables in different forms and combination. Both overplots and crossplots of the results can be produced.

Various run time commands are available to freeze, reinitialize and continue the simulation. Some of them are briefly discussed below:

Ctrl-C; stops the simulation. The values of the constants may be changed (using the SET command). The simulation can then continue via the CONTIN command.

CONTIN; the run may be continued where it left off. This operation bypasses writing the initial condition into the state vector so that the program will continue to integerate from the previous position in state space. For example:

model:

```
    CONSTANT TF= 10.0
    TERMT ( T.GE.TF )
```

run-time commands:

```
    START             ( Run to 10.0 sec )
    SET TF= 15.0      ( Extend to 15.0 sec )
    CONTIN            ( Run 10.0 to 15.0 sec )
```

REINIT; reinitialize the current value of the state variable and writes them back to the initial condition table, thereby destroying the original numbers on the table.

To recover back to an original condition the commands SAVE (store the initial condition table) and RESTOR can be used.

4.2.2 Linearization (ANALYZ Command)

The ANALYZ command invokes a linear analysis capability that can evaluate the Jacobian, trim the state variables to null the rates and also calculate eigenvalues and their associate eigenvectors.

The steady state solver allows the computation of the steady state at less computational cost than using an integration method. The algorithm, however, often fails in non-linear cases. This is illustrated in the following example:

The Jacobian matrix for a non-linear chemical (reaction) system:

$$\dot{X1} = 1 - P1 * X1^{1/2} * X4$$

$$\dot{X2} = P1 * X1^{1/2} * X4 - P2 * X2$$

$$\dot{X3} = P2 * X2 - P3 * X3$$

$$\dot{X4} = P3 * X3 - P4 * X4$$

evaluated at its steady state values (for P1= P2= P3= P4= 1);

A - By hand:

-0.5	0.0	0.0	-1.0
0.5	-1.0	0.0	1.0
0.0	1.0	-1.0	0.0
0.0	0.0	1.0	-1.0

B - Using NAG Routine [4], gives similar results to A (up to four decimal point, using step length h= 1.0E-05).

C - Using ACSL, ANALYZ command (h= 1.0E-04):

0.	0.	0.	-1.4105400
0.	-1.0000000	0.	1.4106700
0.	1.0000000	-1.0000000	0.
0.	0.	1.0000000	-1.0000000

It took the program 15 minutes to evaluate this matrix. With integration step length h= 1.0E-05, the program went into an infinite loop and never recovered.

5. EASE OF USE

5.1 Model and Experiment Definition

To be able to write an ACSL program, the user has to have an understanding of:

1) The ACSL structure and its functions.

2) The FORTRAN programming language.

3) The Editor on the computer.

Obviously, learning all of these is a demanding task, especially on a novice and/or an infrequent user.

The difficulties in using the operators in ACSL has been stressed in [5]. This is partly because there are a large number of operators existing in ACSL. The user has to remember many of them to use the ACSL effectively. The operators also have unfamiliar names which do not reveal their 'functionality' (e.g. REALPL for first order lag).

5.2 Model Amendment

ACSL is not designed as an interactive language, and amendments have to be made to a source file using a system editor. This does not lead to a very fast or very convenient model development.

5.3 Simulation Verification

Although ACSL diagnostics are good, the lack of interactive features as noted above does not aid the verification process.

6. SCOPE FOR IMPROVEMENT

6.1 Model Description Language

To define a model in an interactive and easy to use system, the user needs to remember very little about the system or the description language syntax. A menu driven or a form filling type system presents the user all the necessary model components

and parameter which are to be specified.

Any structure of this sort, however, need to be reasonably efficient in terms of the time required to define a model.

6.2 Editing

Interactive simulation system must permit the user to correct or alter a model before simulation or when simulation has been interrupted. A dialogue type edition may be used, where the user is prompted for the type of model component to be edited. A simple command language may also be added for an experienced user to modify the model rapidly.

6.3 Simulation Monitoring

Separation of model and experiment definitions is becoming very important in simulation languages. The complexity and size of the systems being simulated has grown rapidly in recent years. These systems are usually implemented by experts. To allow the novice users to access such large models, we need:

1) A flexible and interactive experimental frame specification system.

2) Some sort of security system, which protects the model being corrupted or modified by the user.

7. CONCLUSION

ACSL is a powerful simulation language. It provides many robust integration algorithms, extensive simulation function library, Macro, good algebraic loop solver and has reasonable graphics capabilities.

ACSL, as many other existing simulation languages, is not tailored for the need of the people with little or no experience in simulation. It assumes that the user is familiar with computer programming techniques and in particular with FORTRAN. From the discussion given in this report, it is apparent that the areas in which major step need to be taken to improve the user interface are model definition and experiment specification. In summary, the main requirements are:

1) Minimisation of what the user needs to type.

2) Flexible representation of model components. For example, differential equations and transfer functions.

3) Sub-model specification and connection.

4) Provision of user libraries.

5) Provision of a security system.

There are several different approaches which can be taken to produce better simulation software. Writing a new simulation language is, obviously, not a satisfactory solution, as some good facilities (which have been through an evolutionary process in the last two decades) are already available in languages like ACSL.

Designing a 'simulation environment' [6] is a more constructive approach, where a simulation language is integrated with other tools such as a pre-processor, a post-processor and/or an expert system. Together, they form an interactive and intelligent simulation environment. The simulation language will be the heart of this system. Other tools will act as interface between the user and the simulation language. The EASE+ [7] package is one such systems which is already available. It is a pre and post processor for the ACSL and allows programs to be developed graphically. For a more detail discussion of EASE+ facilities see the report 'Evaluation of EASE+ACSL' [8].

ACKNOWLEDGEMENT

The financial support provided by the British Gas for the resarch is gratefully acknowledged.

REFERENCES

1. Mitchell E.E.L. and J.S. Gauthier, 'Advanced Continuous Simulation Language (ACSL)', SIMULATION, March 1976, pp. 72-78.

2. Luker P.A, ' Computer-Assisted Modelling of Continuous Systems', Ph.D. thesis, University of Bradford, 1982.

3. Mitchell and Gauthier, Advanced Continuous Simulation Language (ACSL), User guide/Reference manual, P.O. Box 685, Concord, Mass. 01742.

4. Arabshahi S., 'Linearisation and Eigenvalue Trajectory Plotting of Nonlinear Systems', MSc. Thesis, Dept. of Control Eng., Sheffield University, Sept. 1985.

5. Ekbatani M., 'Evaluation of Advanced Continuous Simulation Language (ACSL)', MEng. Thesis, Dept. of Control Eng., Sheffield University, Sept. 1986.

6. Cellier F. E, 'Simulation Software: Today and Tomorrow ', in Simulation in Engineering Sciences, J. Burger et.al (eds.), IMACS, 1983.

7. EASE+ACSL User Application Manual, Mitchell and Gauthier associates, Inc. and Expert-EASE System Inc., April 1986.

8. Rahbar, M.T., S. Bennett and D.A. Linkens, 'Evaluation of EASE+ACSL', Control Engineering Department, Sheffield University, Research Report No. 305, Sept. 1986.

APPENDIX A

```
PROGRAM EJECTION

INITIAL

      "------------------------DEFINE ALL PRESET VARIABLES "
      CONSTANT  THEDEG = 15.0, DEGRAD = 57.3 ...
          ,MASS = 7.0          , Y1 = 4.0 ...
          ,CD = 1.0            , S = 10.0 ...
          ,G = 32.2            , RO = 0.0023769 ...
          ,VE = 40.0           , VA = 900.0 ...
          ,XMN = -60.0         , YMX = 30.0 ...
          ,TMX = 4.0
      CINTERVAL       CINT = 0.01
      "-----------------------EJECTION ANGLE IN RADIANS"
      THE      = THEDEG/DEGRAD
      "------------------------SEAT INITIAL VELOCITY"
      VX       = VA - VE*SIN(THE)
      VY       = VE*COS(THE)
      VIC      = SQRT(VX**2 + VY**2)
      THIC     = ATAN2(VY, VX)

END $" OF INITIAL "

DYNAMIC

DERIVATIVE

      "------------------------RELATIVE POSITIONS"
      X        = INTEG(V*COS(TH) - VA, 0.0)
      Y        = INTEG(V*SIN(TH), 0.0)
      "------------------------SPACE VELOCITY AND FLIGHT PATH ANGLE"
      V        = INTEG(YGE1*(-D/MASS - G*SIN(TH)), VIC)
      TH       = INTEG(YGE1*(-G*COS(TH)/V), THIC)
      "------------------------COMPUTE DRAG"
      D        = 0.5*RO*CD*S*V**2
      "------------------------USE PROCEDURAL FOR SWITCH TO KEEP SEAT"
      " CONSTRAINED TO GUIDE RAILS. THIS OPERATION IS BETTER DONE BY - "
      "        YGE1 = RSW(Y .GE. Y1, 1.0, 0.0)                 "
      " BUT IS SHOWN HERE TO DEMONSTRATE USE OF A PROCEDURAL BLOCK "
     PROCEDURAL( YGE1= Y, Y1 )
      YGE1     = 1.0
      IF(Y.LT.Y1) YGE1 = 0.0
     END $" OF PROCEDURAL "

END $" OF DERIVATIVE "

      "------------------------SPECIFY TERMINATION CONDITIONS"
      TERMT(X.LE.XMN .OR. Y.GE.YMX .OR. T.GE.TMX)

END $" OF DYNAMIC "

END $" OF PROGRAM "
```