OUTLINE PROPOSALS FOR A NEW STANDARD FOR CONTINUOUS-SYSTEM SIMULATION LANGUAGES (CSSL 81)

Contents

1. INTRODUCTION

This document represents the ideas of a group working in the Simulation Laboratory, Department of Electrical Engineering, University of Salford, regarding a possible replacement for the 1967 SCi Standard Continuous System Simulation Language (CSSL)(1). For a number of years now the opinion has been increasingly expressed that for all its merits the 1967 specification has become steadily more and more out of line with developments in computer languages and in the demands made by users of simulation software.

It would be presumptuous, to say the least, to suppose that an outline specification produced by one group, even with frequent reference to the ideas of other workers, will prove generally acceptable without considerable modification and improvement. Our aim has been, therefore, to present a draft proposal as a basis for discussion and development. Our hope is that the ideas presented here will contain sufficient general appeal to help focus future discussions on a replacement for the 1967 standard. The experience of the authors with a number of Conferences and Committees which have addressed the problem in the past few years has convinced them that unless an "Aunt Sally" or "Strawman" of this kind is produced, progress is likely to be very slow.

At this stage it may be helpful to describe our general approach to producing this draft specification. First of all we have tried to maintain a balance between two apparently contradictory aims. On the one hand we recognise the need to make radical changes in some aspects of the CSSL specification, for example the submodel features and the programming of discontinuous events - on the other hand we have tried as far as is reasonable to maintain continuity. These proposals should therefore be seen as an evolutionary rather than a revolutionary development.

One question which has been debated at some length in recent years is whether the replacement for the CSSL specification should define a combined language with all the features required for continuous, discrete and combined simulation. Our view is that it is necessary first to reach agreement on the requirement for continuous simulation. We do, however, subscribe to the view that the route to combined simulation is eased by the proper representation of discontinuities and hope that by paying careful attention to this important question, we have left the way open to an acceptable standard for combined simulation.

Another consideration which was also borne in mind is that any future standard should incorporate features which facilitate implementation on multiprocessor systems. Regarding the question of implementation, a balanced approach has been sought which will produce a standard uncommitted to any particular method of implementation or type of hardware whilst maintaining an awareness of likely implementation problems.

Finally, few readers will fail to notice that many of the proposals presented here are strongly influenced by trends in modern programming languages and, in particular, Ada (2) and to a lesser extent Pascal.

In conclusion, this document is presented in the form of a discussion of our proposals with a few examples of the style of coding we have in mind, to illustrate the main principles. We are in a position to produce a more formal specification but prefer to await reaction to this document before doing so. A more detailed discussion of many of the ideas underlying the proposals has been presented by Hay (3).

2. GENERAL FEATURES OF THE PROPOSED STANDARD

2.1 Experiment and Model

Attention is confined to continuous system simulation with advanced discontinuity features. The concept of a simulation study, comprising a system model and an experiment to be performed on the model, is strongly supported. Ideally these two aspects of the simulation study should be represented by independent program sections so that a particular experiment might be applied to alternative models or a single model be subject to different experiments.

2.2 Model Structure

It is proposed that the INITIAL/DYNAMIC/TERMINAL structure be retained although some modification is necessary in details. It is further proposed that the concept of derivative, or model definition, sections be retained within the dynamic region to separate code which is executed during the integration calculation from that which is executed only at each communication interval.

Most systems which are the subject of dynamic simulation are represented in the form of a number of interconnected sub-systems. It is proposed that the program defining the model be capable of reflecting this type of structure by means of program modules defining different subsystems. These modules are seen as independent software blocks capable of being built into different systems. To give complete generality subsystems should themselves be capable of being composed of small subsystems although it it appreciated that the extent to which this nesting would be practicable is limited.

With this hierarchical approach to system building the program statements are seen as system primitives from which the subsystem and systems are formed. On this basis it is clear that all the statements used in model definition sections must retain the CSSL practice of defining one or more output variables in terms of one or more input variables.

Given a language of this form it is likely that libraries of models and submodels would be established and these could be linked together as required. This represents an alternative to the use of macros in current languages.

2.3 Representation of discontinuities

A major shortcoming of current CSSLs is in the area of describing and handling discontinuous systems. In recent years numerical techniques have been developed (4)(5)(6) which permit the accurate detection of discontinuities and efficient integration across them. Techniques of this kind are now capable of being invoked automatically from program statements which describe discontinuities of all types naturally and economically.

There are a number of simple types of discontinuity which are sufficiently general to justify inclusion in any simulation language as standard functions. This was recognised in the 1967 specification by the incorporation of limiters, dead space, comparators and switches of different kinds. It is recommended that a similar list of functions be retained in the new specification. These functions do, however, provide only a limited capability for modelling discontinuities of more complex types.

In cases where the library of discontinuous elements proves inadequate the language should provide an appropriate structure to allow users to specify the particular dis-

continuous element required. At least two current languages, COSY (6) and ISIS.80 (7) demonstrate that these features can be built into a CSSL.

2.4 Experimentation with the model

The CSSL 1967 specification defined, by means of the INTERPRETER feature, a means of providing the modeller with some control over the running of his model. This is a feature which has been given increasing emphasis in recent years and modern CSSLs such as, for example, ACSL, provide the user with a range of run-time commands by means of which the model can be run, input data changed, output directives changed and run-time control features (step-length, errors, etc.) changed.

This provision does, however, fall a long way short of the ideal and some languages (e.g. ISIS) have, in addition, included a control section which controls the calling of the model and permits the use of a range of procedural statements including conditional and branching statements. It thus becomes possible to embed the model within a main program capable, for example, of performing iterative sequences of runs such as in parameter optimisation studies.

It is proposed that this feature be included in the specification with effectively the flexibility of a general purpose procedural language (based on Ada constructs in preference to FORTRAN). The model execution could be invoked by a statement similar to the SIM statement in ISIS.

3. OUTLINE SPECIFICATION FOR CSSL 81

Having discussed the general principles, it is appropriate to examine how they may be achieved in practice. The lexical style is first considered which determines how elements of the language appear in a user's simulation program. The program structure and statement detail are then examined.

3.1 The lexical style

The lexical style determines how the various elements of the language (identifiers, numbers, operators) are presented in a users program. It also determines the general presentation or format of the program. The proposed style is based on the Ada specification (2). This does not, however, influence either the syntax or semantics of the language. In fact a FORTRAN-like syntax and meaning (semantics) may still be employed. The original CSSL specification took a similar view with regard to its lexical style.

Ada is a powerful general purpose language which is rich in advanced features. A simulation language has special simulation features but does not need to provide anything like the full range of Ada features. The lexicon specification proposed is a subset of that available for Ada. Some delimiters appropriate for sophisticated computation may be omitted and certain simplifications may be made.

A lexical style based on Ada has the following major advantages over competing styles.

a) Programs using the Ada style are well structured, readable and self-documenting (to a certain extent), and it imposes sensible rules on allowable program statements and structures. There is also an underlying trend towards this style by computer users.

b) The program presentation allows efficient conversion processes to obtain an equivalent program in an executable form.

The objectives of a general purpose language such as Ada and a language for the specific task of simulation are different. Although our proposal will always default to the Ada style it should be clear that where there is conflict the simulation requirements will override other considerations.

The examples given later in this paper are written in the proposed style.

3.2 Simulation program structure - the Experiment

It has already been argued that the experiment and the model should be separated. The program code to define a simulation experiment may be expressed in most general purpose scientific computer languages such as FORTRAN or Ada. Only minor extensions to a general purpose scientific language code, or procedural code, are required to adequately describe a simulation experiment. Note that the experiment is a sequential type of operation as opposed to the actual simulation solution which represents parallel processes.

It is proposed, therefore, that the experiment should be a complete program module, that is a unit similar to a

FORTRAN main program or subroutine. To achieve portability the form of the program module will be specified by the new standard! The form of the experiment module will be a considerably reduced subset of Ada. The subset chosen will allow conversion or translation to an equivalent FORTRAN module as well as conversion to an Ada program. The data types will be restricted to those permitted in the model description section of the program, for example, single or array variables of type real, integer, logical and enumeration. It is accepted that other data types such as 'character', double precision, complex and records may be regarded as extensions to the specification..

The following program example illustrates the form of the proposed experiment module.

```
--    comment on start of declarations
      DONE: logical (false); -- initial value given to DONE
      PAR, NEW_PAR: real;
      RESULT: real;
      DELTA: constant real (0.01);
--    declare interface to external library routine
      procedure OPTIMISE_SV (FLAG: out logical; NEXT_X:
                  out real; OF, X, DX: in real) external;
--    experiment start
         PAR:=  0.5;
         while   not DONE
         loop
            -- invoke simulation
            solve PLANT (RESULT:=  PAR);
            --  invoke mathematical library optimiser
            OPTIMISE_SV (DONE, NEW_PAR, RESULT, PAR, DELTA);
            PAR: = NEW_PAR;
         end loop;
--    print result
         print "Result =", RESULT," for parameter = ", PAR;
      end experiment;
```

This program experiment repeatedly solves the simulation model identified by PLANT. The only direct communication between experiment and model is through the model input variable PAR and the model output variable RESULT. A mathematical library routine is used to predict values for PAR which will lead to an optimised value for RESULT.

The experiment expressed in a sequential general purpose scientific language style controls the solution (simulation) of one or more models. Communication of data between the experiment and the models is achieved by the model argument list and also by means of a pre-defined simulation environment. This environment means that certain reserved variables are assumed to be declared in both the experiment and the model. The reserved variables include: the dependent variable of integration (T), communication interval (CINT), end of run value of the independent variable (TFIN) and other variables describing the integration process. Users may accept the default values for reserved variables or override the defaults either in the model or in the experiment. Provision would also be made to rename reserved variables to suit a particular problem, for example the declaration:

```
X: new T;
```

makes X the independent variable instead of T.
Note that this approach does not preclude the possibility of employing different integration methods in the same simulation (see section on model segments).

3.3 Simulation program structure - the Model

The model has the job of describing a physical system and information relating to how a single simulation run is to be performed. Multiple runs may, of course, be invoked from the experiment.

The main structure of the model follows the ideas expressed in the original CSSL specification in that there is an optional INITIAL region, always a DYNAMIC region, and an optional TERMINAL region. Pre-run initialisation is undertaken in the INITIAL region; the DYNAMIC region describes the parallel processes of a physical system and information relating to the control of the simulation; and the TERMINAL region allows end-of-run calculations to be performed either to produce output or to calculate results which are to be returned to the experiment. Note that control may only pass from the TERMINAL region to the calling experiment. Control may not pass directly from the TERMINAL to the INITIAL region. A further model may be invoked from all regions except the model definition section of the DYNAMIC region.

The model argument list is separated into two parts: an output and input list. Variables that appear in the input list communicate data from the experiment to the model, and variables in the output list communicate end-of-run information to the experiment. An input list argument is regarded as a constant from the model's point-of-view and its' value may not be changed. An output list variable is considered to have no value (may not be used) until its value is defined in the model. Note that for simulation models (unlike Ada) an argument may not normally act as both an input and an output argument.

Further information may be passed from the experiment to model by the predefined simulation environment. The following example shows how a model uses the predefined simulation environment.

```
MODEL       PLANT (RESULT:=PAR);
            A,XFINAL: constant real (36.0, 100.0);
            X: real (0.0);
INITIAL
            RESET; -- restores conditions at start of
                   --previous run
            CINT:= 0.25;
            ALGO:= FS_RKA;
            TFIN:= 2.5;
DYNAMIC
            X'' := A* (XFINAL-X)-PAR*X';
            RESULT:= INTGR(T*(XFINAL-X)**2,0.0);
Communication
            TERMINATE RESULT > 1000.0;
            PLOT T,X, 0.0, TFIN, 0.0, 160.0
            PREPARE T,X,X', RESULT;
TERMINAL
            if  T < TFIN then
                 print "result large run aborted";
                 end if;

end;
```

3.3.1 Declarations

The model requires all identifiers to be declared, but a pre-defined declaration of simulation variables is the default. For example, T, CINT, ALGO, TFIN and FS_RK4 are all assumed to be declared in the predefined environment and therefore do not require re-definition in the model. The variable T takes the default value of zero. All other

variables, however, require explicit declaration.

The types of the model arguments have defaulted to type real. If the RESULT had been of type integer then the first statement could be:

MODEL PLANT (RESULT: integer:= PAR:real)

The aggregate (36.0, 100.0) following the introduction of A and XFINAL gives the constant values of these variables. The aggregate (0.0) following the introduction of X gives an initial value for X which is restored on each model invocation.

Arrays of the basic types could have been introduced by declarations of the following forms:

```
VECTOR: array (1.. 10) of real;
MATRIX: array (1..2, 1.. 3) of integer;
TABLE:  array (1..4) of real (2.0, 4.0, 8.0, 16.0);
DY_ARR: array (1..N) of real; -- where N is an argument
```

Arguments may also be declared as arrays, e.g. MODEL PLANT (RESULT: array(*) of real: = PAR). The * means that the dimension of the array is inherited from the calling module.

It is appropriate to note that variables used in a model fall into a small number of classes which are:

(a) Constants which do not change values at any time. Model input arguments are regarded as constants during the model execution.

(b) Parameters which remain constant during the actual simulation, that is during the period the DYNAMIC region is being processed. These variables are given values in the INITIAL region.

(c) Model variables (algebraic) should only be given values in the DYNAMIC region. These variables represent the output of a block representing a part of the physical system. The values for algebraic model variables require calculations unlike history variables.

(d) Model variables (history) are similar to their algebraic counterparts but their current values depend on the past (history). An integrator or delay function output is a history variable. Unlike the algebraic variables it is appropriate to give these variables initial values in the INITIAL region.

(e) Output variables only appear in the terminal region and are used in connection with direct output (to graph, file, printer etc.) from the TERMINAL region or are output arguments of the model.

Note that model output arguments may only be in classes (c), (d) and (e) and input arguments in class (a). This indicates the class (b) parameters are always local to the model. A model argument output variable of class (d) (history) may act as both input and output variable from the experiment's point of view.

3.3.4 INITIAL region

The INITIAL region is expressed in procedural code enhanced by a small number of special simulation statements such as

RESET. The purpose of the region is to set values for parameters, give initial values to history variables and possibly produce some form of pre-run output. At the physical end of the initial region the dynamic region is automatically entered and the simulation run is started.

3.3.3 TERMINAL region

Following a run the procedural code of the TERMINAL region sets output variables and produces output before passing control back to the calling experiment.

3.3.4 DYNAMIC region

The DYNAMIC region unlike the other regions is not expressed in procedural code which is appropriate for the sequential operations performed in the experiment, INITIAL and TERMINAL regions. The DYNAMIC region describes the parallel processes which represent the physical system being simulated. The region is divided into two main parts the model definition section and the communication section.

Model definition section: the model definition section has the task of defining the set of model variables of class algebraic or history. No other variable may be defined in this section. All algebraic and history variables must be defined by a single statement (which may be a compound 'if' or 'when' statement) in the section and may not be defined by more than one statement. Each statement of the section defines a 'block' which may be regarded as a physical element of the real system being simulated. Therefore each statement defines one or more outputs in terms of one or more inputs.

Submodel invocation: A physically identifiable system block may be treated as a separate entity in the simulation as well as being physically separate from the remainder of the real system being simulated. Such a block may be represented by a submodel! For example, it may be appropriate to represent the controller of a machine by a submodel and, of course, if desired the machine itself by another submodel. A submodel has a similar structure to a model, and an experiment-model relationship is similar to a model-submodel relationship. The latter is, however, more intimate in that information is communicated between model and submodel during each execution of the model definition section of the DYNAMIC region.

A spin-off from this approach is that a collection, or library, of submodels may be created and freely used in different simulations. This demands that a submodel does not depend on the environment from which it is invoked except by information communicated by an argument list or by the pre-defined simulation environment. It should also be made clear that if a submodel is invoked more than once in the model definition section that separate identical subsystems are being represented. In these cases no conflict arises between the separate invocations of the same submodel. The submodel concept replaces the need for the MACRO of the original CSSL specification.

A submodel representing a subsystem CONTROLLER with outputs Y1 and Y2 and inputs, X1, X2, X3 and X4 may be invoked by a statement of the form:

```
CONTROLLER (Y1, Y2:=  X1, X2, X3, X4);
```

A subsystem LAG with a single output Y1 and two inputs X1 and X2 may be invoked by either:

```
LAG (Y1:= - X1, X2);
```

or by

```
Y1:= - LAG (X1, X2);
```

This second form of invocation could be embedded in an arithmetic expression. Standard submodels are provided and the example model given earlier used the INTGR standard submodel.

Sorting: The model definition section is computed, possibly several times, before the integration system is able to produce satisfactory solutions for a new value of the independent variable. Each computation of the section is known as a DYNAMIC pass. Statements of the section may be automatically sorted so that they are executed in a sequential order which ensures that all inputs to a block are known prior to the block code computation. During this sorting process advantage is taken of the property of history variables that their values are known, without computation, at the start of the DYNAMIC pass.

Communication Section: The communication section is automatically executed at the initial value of the independent variable (T), and after the integration system has produced solutions at values of independent variable which have changed by an amount equal to the communication interval (CINT). That is, the communication section is invoked at communication points which occur following each integration period of CINT.

This section has the function of allowing information interchange between the simulation process and the outside world, and also imposing some control over the simulation. In

the case of parallel simulation segments (see later) information is also exchanged between the segments.

The statements in the communication section are procedural language statements with the addition of a number of simulation operations. The following indicates some of the simulation operations which are available:

TABULATE T, RESULT, X;
produces a heading at the start of a run and then outputs T, RESULT and X to the terminal or nominated device at each subsequent communication point.

PREPARE T, X, X';
saves values of T,X and X' at each communication point for subsequent analysis or plotting.

PLOT T, X, 0.0, TFIN, 0.0, 160.0;
produces a graph of X against T as the simulation proceeds.

TERMINATE RESULT>1000.0
causes the simulation to terminate when the condition becomes true and for control to be passed to the terminal region. A default TERMINATE statement of the form

TERMINATE T>=TFIN
is always active

Of the above statements only PREPARE could appear in another program section.

Data may be read from a file in this section to set a simulation parameter. Such a parameter would remain constant until the next communication point where it could be updated by further information from the file.

3.4 Advanced language features

This description shows how the basic simulation features already described are extended to provide advanced features.

3.4.1 Parallel Simulation Segments

Segments are introduced to provide facilities to employ multiple parallel processors to simulate a physical system. The same mechanisms allow real hardware, such as an electronic controller, to form part of the simulation. In addition simulation on a conventional sequential computer can employ different integration algorithms for the solution of different parts of the real system.

The following partial simulation model shows how parallel segments are invoked.

```
DYNAMIC
   process CONTROL (FIELD:=SPEED,REFERENCE);
   process MACHINE (SPEED:=FIELD);

communication
   .......
```

This example shows that a machine and its control are simulated separately and only interchange information at communication points. The CONTROL segment could be real electronic hardware, or, in fact, the machine could be a physical machine. Where the segments are simulations rather than actual physical hardware the definition of a segment is similar to that of a model.

The definition of a segment only differs from a model in the following respects:

(a) The keyword MODEL is replaced by SEGMENT.

(b) There is no TERMINAL region in a segment.

(c) The pre-defined simulation environment for a segment differs from that for a model.

In a segment the following predefined variables are accessible but may not be changed from within the segment:

T, CINT, TFIN

It is the responsibility of only the controlling model to set and act upon these variables.

Each segment has an independent set of other reserved variables which allow the segment to specify its own integration method and associated integration control. Such integration must be capable of synchronisation with the calling model's CINT specification.

Where segments are executed on a conventional sequential computer it does not matter which segment is computed first. Each segment integrates its equations for a period of CINT and then, and only then, is information exchanged through the segment arguments.

A model definition section which contains a segment invocation cannot contain any other statements except further segment invocations.

3.4.2 The Submodel definition

A submodel may be invoked from the model definition section of a model, the same section of a segment or another sub-

model. The form of a submodel specification is similar to a segment with the following differences:-

(a) The keyword SEGMENT is replaced by SUBMODEL.

(b) The submodel has access to the pre-defined simulation environment but it may not change any of the reserved variables. It has read-only access to reserved variables.

(c) A segment may only be invoked once by a model whilst the same submodel may be invoked several times from a model, segment or other submodel.

The relationship of a submodel to its calling model or segment is more intimate in that information is interchanged during each DYNAMIC pass.

As segments may be executed on different processors the variables which are local to a segment are clearly private to the segment and the processor on which they are computed. The same rule applies to submodels in that local variables are private to a particular invocation of the submodel. For example, if a submodel is invoked twice and it defines and uses a local variable Q then the storage unit associated with Q for the first invocation is a separate storage unit from that used for Q in the second invocation.

In the original CSSL specification MACROs provided users with a feature which was, in some respects, similar to the submodel. The specification demanded that a macro-text expansion mechanism be used. We, however, do not force such an implementation nor, on-the-other-hand, preclude the possibility of implementing submodels using text substitution macros. It should be noted that Ada procedures may either be treated as subroutines in the FORTRAN sense

or expanded 'macro style'. The same definition of the procedure is employed independent of the implementation route.

3.4.3 Discontinuities

Discontinuity statements may appear in the model definition sections of models, segments or submodels. These statements permit discontinuities to be accurately and efficiently modelled (4,5) in an unambiguous fashion. Common discontinuous elements will be represented by standard submodels available in a library of special simulation functions.

Basic discontinuities: To illustrate how such discontinuous elements are represented let us consider the example of a simple limiter in which the output Y is defined as:

```
Y = UL when X > UL
Y = LL when X < LL
Y = X at other times
```

This limiter may be represented by the following program statement.

```
Y:= if X > UL then UL
    else if X < LL then LL
    else X;
```

The discontinuity detection mechanisms which work in conjunction with the integration process detect when the relational conditions (say X > UL) become true or false. The detection, however, pin-points when the relation changes state (say false to true) to within a certain error bound. For example, the default error bound of 0.001 means that X may be as large as UL + 0.001 at the point where

the relation X > UL is detected as becoming true. It is because of this error bound that the relational operators equal and not equal are not allowed. The allowable operators are: >, <, >= <=.

The user may adjust the error bound to suit a particular problem by using the following extended form of discontinuity statement.

```
Y: = if  X >/0.01,  0.005 / UL then UL
     else if  X </0.0001 / LL  then LL
     else  X;
```

This means that:

X > UL is detected as becoming true when X changes from a value where X ≤ UL to a value in the range

UL < X < (UL + 0.01).

It is detected as becoming false when X changes from a value where X > UL to a value in the range

UL ≥ X > (UL - 0.005)

The formulation of the equations for the above limiter could cause problems in rare cases due to the fact that there is a small (less than the error band) instantaneous change in the output Y as it becomes limited. For example, as the device goes into the limited state there may be an instantaneous change from, say, Y = UL + 0.0057 to Y = UL. This problem may be overcome by a reformulation of the limiter as follows:

```
Y: = if  X  < =  UL  and  X > =  LL then X;
```

This form of statement allows Y to retain the value it had at point the discontinuity was detected and the device switched to the limited state. To avoid the problems associated with starting a simulation with the device in the limited state it is necessary to initialise Y in the INITIAL region. The need to have such an initialising statement in the INITIAL region which is separated from the discontinuity statement in the DYNAMIC region can be regarded as a nuisance.

Using submodels to represent discontinuous elements allows the latter problem to be overcome in the following manner:

```
SUBMODEL LIMIT (Y:=X, LL, UL);
INITIAL
   if X > UL then Y:= UL
   else if  X < LL  then Y:= LL
   else Y:=  X;
DYNAMIC
   Y:=  if X < = UL and X > = LL then X;
END;
```

This submodel could be invoked by:

```
OUT:=  LIMIT (IN, LOW, HIGH);
RESULT:=  INTGR (LIMIT(IN1, LOW1, HIGH1), 0.0);
LIMIT (OUT2:= IN2, LOW2, HIGH2);
```

Note that there may be any number of invocations of the same submodel!

Discontinuity triggers: A second form of discontinuity description allows the change to trigger a certain action. Consider the example of a comparator where:

```
Y  =  1.0 if  X > limit
Y  =  0.0 otherwise
```

This could be programmed using the 'if-assignment' form, that is

```
Y:=  if  Y > limit then 1.0 else  0.0;
```

This comparator has the characteristic that its output only changes when the discontinuity occurs, that is, when the relation Y > limit either becomes true or becomes false. The following (illegal) form of an 'if' assignment gives an appropriate description.

```
relation:=  X > limit;
Y:= if relation changes to true then 1
else
if relation changes to false then 0;
```

The 'when' assignment form is provided to do this task.

```
Y:=  when  X > limit then 1 else 0;
```

This sets Y only when a change occurs in the relation X > limit. The form:

```
when relation then
```

is interpreted as 'when the relation becomes true' and the else clause as 'when the relation becomes false'. If there is no change in the relation then no operation is performed.

A function to calculate the approximate rate of change of a variable illustrates the need for a further extension.

Consider

```
RATE = (X-Xlast)/(T-Tlast)
```

There must be a mechanism to allow Xlast and Tlast to be updated during a DYNAMIC pass which follows the completion of an integration step. That is, when results are valid and the pass is not an iterative pass which is being used by the integration system as part of its calculations to produce the next valid (end-of-step) results. The following statement form is proposed which uses a 'when' assignment formation and a reserved logical variable VALID_STEP to determine whether action is required.

```
Ylast:= when VALID_STEP then X;
Tlast:= when VALID_STEP then T;
```

(Note that an 'if' instead of 'when' could be used although the 'when' shows the transient nature of VALID_STEP).

A more compact way of expressing the last two statements would be:

```
when VALID_STEP then
                    Xlast:=  Y;
                    Tlast:= T;
                end when;
```

This form or the original form, of statement do not require sorting because they only have the job of updating history variables Xlast and Tlast. History variables have correct values on every entry to the DYNAMIC pass. The above statement implies that, prior to the next pass, the updating must take place, and it does not mean that Ylast

is updated during the actual DYNAMIC pass.

An 'else when' and an 'else' clause are acceptable extensions to the 'when' statement.

A normally difficult to program hysteresis function is easily specified with the new statement forms.

```
Y:= if X > Ylast + UL then X - UL
    else if X  < Ylast + LL then X-LL
    else Ylast;

Ylast:= when VALID_STEP  then Y;
```

Non assignment conditional statements

The simple 'if' or 'when' assignment form of statement is suitable for many discontinuous functions but is somewhat cumbersome for more complex types. In these cases the statement form of 'if' or 'when' is more appropriate. Consider code to produce a pulse train (1 or 0) in which the output is 1.0 (on) for 'Ton' time and then 0.0 (off) for 'Toff' time.

```
when T ≥ Tnext then if  Y = 0.0 then Tnext:=T+Ton;
                                     Y:= 1.0;
                                else Tnext:=T+Toff;
                                     Y:= 0.0;
                                end if;
                end when;
```

The statements included in both the 'if' or 'when' compound statements are procedural statements. This gives considerable flexibility for the users to construct their own models of any complexity.

4. CONCLUSIONS

This paper has attempted to provide a fairly detailed outline specification for a new standard for continuous systems simulation languages (CSSL 81). There are still many areas of detail which have not been fully discussed. In fact, when the implementation study has been completed the authors believe it may be necessary (or advisable) to add further features.

Further mandatory declarations are candidates for inclusion on the basis that (a) it is good programming practice and (b) that the translator can be simplified and made more efficient. Diagnostic features have not been discussed but it has been assumed that a CSSL should provide, at least, as good diagnostic and trace features as modern FORTRAN systems (see DEC's VAX or PRIME's FORTRAN aids). The extent to which Ada should be adopted as the procedural language has been left as an open question. This paper has taken the view that only those features that are necessary should be included. For example, the authors concede that Ada procedures, with Ada scope rules, may well have a part to play in a new CSSL specification. Clearly further discussion is required to determine the subset of Ada which is most appropriate for a CSSL. This decision and the formulation of a strict and complete specification is extremely important if the language is to prove acceptable to users, practical to implementors, and is sufficiently standard to provide portability for simulation programs.

This paper has concentrated discussion on the concept of building simulation programs from the sound foundations provided by : models, segments, submodels, strict model definition, comprehensive discontinuity facilities and the ability to use standard libraries of mathematical functions and subprograms.

5. ACKNOWLEDGEMENTS

The authors wish to acknowledge the contribution to the formulation of the above proposals made by colleagues at the University of Salford. The authors also wish to acknowledge the value of discussion and encouragement given by J.G. Ferrante previously of the European Space Agency, ESTEC, Noordwijk, The Netherlands, and now with MATRA, Toulouse. Some of the work on which this report is based was carried out under European Space Agency Contract and their support is gratefully acknowledged.

6. REFERENCES

1) 'The SCi Continuous System Simulation Language (CSSL)' Simulation, Vol.9, No.6, Dec. 1967.

2) 'Preliminary Ada Reference Manual' Sigplan Notices, Vol.14,No. 6, June 1979.

3) Hay, J.L., 'A new CSSL Standard - An implementation view', UKSC Conference on Computer Simulation, Harrogate, 1981.

4) Hay, J.L., Crosbie, R.E. and Chaplin, R.I.:'Integration subroutines for systems with discontinuities' Computer Journal, Vol.17, No.3, 1974.

5) Hay, J.L. and Griffin, A.W.J.: 'Simulation of discontinuous dynamical systems', SIMULATION OF SYSTEMS '79 Ed. L. Dekker et al, North Holland Pub. Co., 1980.

6) Cellier, F.E.:'Combined continuous/discrete system simulation by use of digital computers', Doctural dissertation, ETH Zurich, 1979.

7) 'ISIS.80 User Manual', Simulation Systems, The Gables, North End, Yatton, Avon, 1980.