MODELING, IDENTIFICATION AND CONTROL, 1986, VOL. 6, NO. 4, 181–199

Evolution and perspectives of simulation languages following the CSSL standard†

MAGNUS RIMVALL‡ and FRANÇOIS CELLIER§

Keywords: *digital simulation, continuous simulation languages, CSSL'67, combined continuous/discrete simulation, standardization.*

Since its publication in 1967 the CSSL specification has served as a successful standard for continuous simulation languages. However, recent advances in modeling techniques, simulation methodologies and computer languages has motivated the simulation community to spur several standardization committees. In this paper the 1967 standard as well as several modern CSSL languages are reviewed, some current trends and developments are discussed and the status of present standardization efforts is presented. An outlook on some of the facilities we can expect to find in future simulation packages (with or without standardization) is given.

1. Introduction

During the early sixties several dozen software products were developed for the digital simulation of continuous dynamic systems (see Clancy and Fineberg, 1965). Many of these products were coded in assembler or contained non-standard language elements making them non-portable. Moreover, as each product used its own model description syntax without standardized building blocks, developed models were not transportable from one simulation language to another and programmers switching to a new simulation environment needed a very long familiarization period. For these reasons, the Simulation Councils, Inc. (SCi) in 1965 decided to form a standardization committee for continuous simulation languages. In 1967, as result of this standardization effort, the specification of the Continuous Systems Simulation Language (hereafter referred to as CSSL'67) was published by Augustin *et al.* (1967).

On one hand, CSSL'67 was a synthesis of elements and concepts from hitherto developed simulation products like MIMIC (Peterson and Sansom, 1965), DSL/90 (Syn and Linebarger, 1966) and MIDAS (Harnett, Sansom and Warshawsky, 1964). On the other hand, the committee deliberately limited the scope of the standard in order to increase its life-span and preserve software portability:

CSSL'67 defined a general modeling environment, which insured a fair portability of models from one CSSL-product to another (standardized user interface).

CSSL'67 is an open-ended standard, new functions and operators may be added to any CSSL language (extendable modeling interface).

Received 15 June 1985.

† This paper was presented at the International Seminar on Modern Methods in Dynamic Simulation of Industrial Processes, Trondheim, Norway, May 1985.

‡ Institute for Automatic Control, Swiss Federal Institute of Technology (ETH), Zurich, Switzerland.

§ Department of Electrical and Computer Engineering, The University of Arizona, Tucson, Arizona, U.S.A.

The committee refrained from formulating specifications on the run-time system. This enabled the implementation of CSSL languages on most kinds of computers using different implementation languages (open computer interface).

Because of the great divergence of simulation software, the need for a standard was imminent in 1967 and soon after commercial products using the new standard emerged on the market (e.g. ACSL and CSSL-IV). Thanks to the expertise and farsightedness of the standardization team, CSSL'67 continued to serve as a template and yardstick for the development and evaluation of simulation languages for more than a decade. However, rapid technological developments in software engineering, modeling techniques, numerical integration methods, etc. have during the last years resulted in a divergence of the existing packages from CSSL'67. Currently several efforts to establish a new language standard are made (e.g. TC-3 of IMACS and CSSL of SCi). Unfortunately, the situation today is radically different from 1967; more successful commercial products with their own 'standards' have established themselves and the individual members of the now much larger simulation community all want *their* particular needs to be fulfilled. All this makes the standardization work, plus any subsequent enforcement, much harder and so far no acceptable new standard has emerged. Nevertheless, a consensus seems to exist on some of the facilities to be included in the new standard. These features include better modeling support (including hierarchical modeling with flexible submodel interconnection), discontinuity handling and more versatile experimental frames. Other fields where standards, at least at the moment, seem to be harder to obtain are the inclusion of discrete elements (discrete events and discrete processes), facilities for graphical model definition, simulation data bases as well as support for parallel computing and real time simulation. Moreover, a standard 'simulation operating system' (much like the proposed Ada language environment) could make the user interfaces totally independent of the implementation machine.

2. Review of CSSL'67

This chapter will give the reader unfamiliar with CSSL'67 an introduction to the standard and prepare him for the following chapters. For a more formal description of CSSL'67, we refer to the standard itself (Augustin *et al.* 1967).

The CSSL'67 standard was designed for the simulation of continuous dynamic systems described by ordinary differential equations. This gave CSSL'67 a general scope compared to many other simulation systems, which were (and are) designed for the simulation of special kinds of models and thereby using bond-diagrams (Granda, 1983), elements of analog computers (Harnett *et al.* 1964), building-blocks depicting electronic elements (e.g. SPICE-II) etc. Consequently, CSSL'67 was to cover the simulation of technical as well as non-technical systems (e.g. mechanical, electrical, biological, economical and political systems). However, the limitation to continuous, lumped systems precluded its use for the simulation of systems described by partial differential equations or models containing discrete elements (see below).

2.1. Historic background

When reviewing CSSL'67 we must bear in mind that the simulation world of the sixties was quite different from today. Most simulations were still performed on

analog computers and the computing power of the existing digital computers was very limited, making hybrid simulations strong alternatives to pure digital simulations.

It is interesting to note how the advocates of digital simulations in the sixties (including the fathers of CSSL'67) always explained how easy it was to transform systems of analog building-blocks into their digital counterpart, whereas any competitive hybrid simulation system of today must be able to automatically transform digital programs (using languages similar to CSSL'67) to hardware-connections in the analog computer!!!

In this perspective, it is remarkable how well the CSSL'67 standard has survived the last two decades. This can only be attributed to the farsightedness of the 'founding fathers' and the extendability and open-endedness of the standard.

2.2. Introductory example

To illustrate the basic modeling capability of CSSL'67, we consider a small ecological system consisting of plants and herbivores living in an encapsulated environment. A simple model of this system would contain two differential equations:

$$\frac{dP}{dt} = K1 \,.\, P - K2 \,.\, P \,.\, P - K3 \,.\, P \,.\, H$$

$$\frac{dH}{dt} = B \,.\, K3 \,.\, P \,.\, H - K4 \,.\, H - K5 \,.\, H \,.\, H$$

where P is the concentration of plants (Kcal/cubic m), and H the concentration of herbivores in the system. $K1 \ldots K5$ and B are known constants, $K1$ and $K2$ indicate the reproduction rate and crowding factor of the plants, $K3$ the grazing effect, and $K4$ and $K5$ the death rate and crowding factor of the herbivores. We note that the concentrations of the two species are interdependent over the term $K3 \,.\, P \,.\, H$. A CSSL-program describing this model would be fairly simple:

```
COMMENT ECOLOGICAL SYSTEM
COMMENT
COMMENT PARAMETERS AND INITIAL CONDITIONS
COMMENT
  DATA [K1 = 1·1; K2 = 1 . E-5; K3 = 1 . E-3; K4 = 0·9; K5 = 1 . E-4]
  DATA [B = 0·02; P0 = 10·0; H0 = 100·0]
COMMENT
COMMENT DIFFERENTIAL EQUATIONS
COMMENT
  P  = INTEG[PD, P0]
  H  = INTEG[HD, H0]
  PD = P*(K1–K2*P–K3*H)
  HD = K*(B*K3*P–K4–K5*H)
COMMENT
COMMENT INFORMATION FOR THE SIMULATION
COMMENT
  TERMINATE [T > 30·0]
  CINTERVAL [,0·6]
  PRINT [T, P, H, 'OVER']
END
```

```
COMMENT
COMMENT "RUN-TIME" CONTROL COMMANDS
COMMENT
GO;
P0 = 100·0; H0 = 10·0;
GO;
STOP;
COMMENT
COMMENT END OF COMPLETE CSSL'67 PROGRAM
```

We note how the differential equations are described using a very natural notation and that the equations can be stated in any order (*PD* and *HD* are used before they are defined); the system will sort all equations into correct order before execution. Most parameters defined in the DATA statement remain constant during the simulation, we only choose to change some of these values in the "run-time commands" to obtain results from two different systems (different initial concentrations). As foreseen already in CSSL'67 (!), most modern CSSL languages allow the user to enter these run-time commands interactively during the execution of the simulation. In later chapters we will see how this can be considered as a first step towards a flexible experimental frame.

2.3. Design goals for CSSL

The previous example very nicely illustrates the first of three major design goals of the CSSL standardization committee:

> To provide a simple and obvious programming tool for the novice user.
>
> To give the sophisticated user a flexible tool with the power needed for the modeling of larger and more complex systems.
>
> In anticipation of future technological advances (e.g. in the fields of graphical displays, interactive computer systems), the system must provide for a flexible expansion.

To meet the first goal, CSSL'67 featured a comprehensive syntax for the description of differential equations, block-oriented representation, preprogrammed and 'user-hidden' sections for the integration and run-time control and intelligible error messages. Furthermore, facilities like sorting and simple and standardized input/output commands were included.

As a consequence of the second goal, the in 1967 widely used scheme of translating the model description into an established procedural language (in 1967 FORTRAN, PL/1 or ALGOL, today FORTRAN, PASCAL or Ada) was retained (Fig. 1). This preprocessing enables the user to include code of the target language (e.g. FORTRAN) in his model description and thereby greatly enhances the capabilities of the simulation language without cluttering it with standard programming elements. Moreover, this translation into a conventional computer language allows the user as well as the language developer to add new operators (functions) to the library, making the system functionally open-ended. The main drawback of this scheme is that the full compilation/link sequence has to be performed each time a change to the model is made. Other simulation languages trade this compilation time against execution time by directly interpreting the model description (e.g. SIMNON of Elmqvist, 1975 and 1977, or DESCTOP of Korn, 1985).

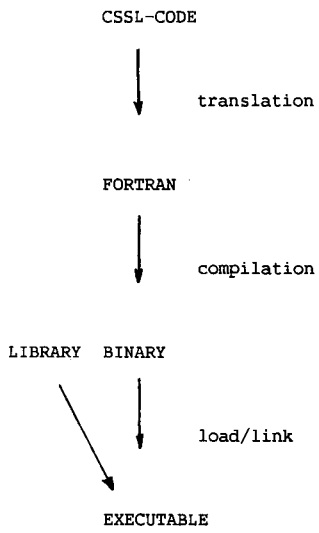

Figure 1. Translation of model description.

The last of the three design goals, the open design of CSSL'67, was in the long run probably the most important one. As we will see in later chapters, it enabled the inclusion of many new features into CSSL languages. This prolonged the period over which CSSL'67 could be used as a standard but also spurred the various CSSL dialects now making a new standardization so hard.

2.4. Structural elements in CSSL'67

Long before terms such as 'structured programming' or 'data structures' became cliches attached to all software products, CSSL'67 defined language elements for the structuring as well as execution of the model. Starting from the hierarchically lowest level, these elements are:

Elements controlling the sorting. As we have seen in the introductory example, the differential equations describing the dynamics of the system are sorted into the correct order for sequential, rather than 'real-world parallel', processing. To enable the inclusion of dynamic parts only sortable en bloc, NOSORT and PROCEDURAL elements can be used:

```
PROCEDURAL [Y = T, T0]
  COMMENT STEP-FUNCTION STARTING AT TIME T0
  Y = 0·0
  IF (T > T0)Y = 1·0
END
```

To avoid repetitive code, *macro* definitions and macro instantiations should be utilized:

```
MACRO TWOINT [OUT = IN, IC, ICD]
  COMMENT ...
  COMMENT Implements OUT = IN, OUT(T = 0) = IC, OUT(T = 0) = ICD
  REDEFINE OUTD
  OUTD = INTEG [IN, ICD]
  OUT = INTEG [OUTD, IC]
```

```
END
...
...
HPOS = TWOINT [HFORCE/MASS, 0·0, 0·0]
VPOS = TWOINT [–G/MASS, ALT, 0·0]
```

Here we define a second order differential operator as macro and use it twice for the horizontal and vertical movement of a body.

The dynamic model can be divided into several *derivative sections*. These sections are particularly useful when the language permits the use of different integration steps/methods for each section, but can also be used for documentation purposes.

In our introductory example a simulation run was completely described through the differential equation and a condition for terminating the simulation. In more complex simulations, the model has to be initialized before each run (setup) and/or results must be evaluated after each run. For this purpose, CSSL'67 supports the initial, dynamic and terminal *regions*:

```
PROGRAM
COMMENT EXAMPLE SHOWING THE CSSL'67 REGIONS
INITIAL
  COMMENT ACTION TO BE TAKEN BEFORE EVERY RUN
END
DYNAMIC
  COMMENT DESCRIPTION OF THE DYNAMICS OF THE SYSTEM
END
TERMINAL
  COMMENT ACTION TO BE PERFORMED AFTER EACH RUN
  COMMENT A RETURN TO THE INITIAL SECTION FOR A NEW
  COMMENT RUN IS POSSIBLE (E.G. FOR OPTIMIZATIONS)
 END
END
```

A few CSSL's allow complete model descriptions to be collected into *segments*, allowing for the simulation over several independent variables (for example over space in addition to time).

On the *execution* level the simulation models are invoked (run). The execution can be described either by a program, by 'run-time commands' or by a combination of both. Using the program approach, a main program would be written in the target language and linked to the simulation package. From this main program the simulation runs would be controlled. Using the run-time approach, commands would be entered to the 'run-time' system interpreting the commands and controlling the simulation, thus allowing for interactive as well as batch versions of CSSL languages.

Although CSSL languages are functionally open-ended, these structural elements circumscribe the extendability and thereby the modeling power of any CSSL. It is therefore not surprising that much of the work on new standards circle around structural issues (in the model as well as execution description) rather than functional issues.

2.5. Functional elements of CSSL'67

The CSSL'67 standard recommends a minimal set of functions and operators to be included in all languages. These operators include:

Integrational operators. Minimal requirement: an open, a limited and a mode-controlled integrator.

A *derivative* operator.

An *implicit function* for the iterative solving of algebraic loops.

A *delay* operator.

Non-linear functions like hysteresis, flip-flops, limiters, table-driven multidimensional functions (interpolated).

Structures should be provided for defining dimensional structures (arrays), constants and other auxiliary variables. Statements controlling the integration (integration method, step size, error limits) and the execution (termination condition, communication interval, variables to be saved as output) must be available.

Once again it should be noted that CSSL'67 is intentionally restrictive in the recommendations for functional elements, as not to hamper the open-endedness of the system.

3. Current CSSL languages

In this chapter we will survey some of the most widely used simulation languages belonging to the CSSL family. Thereby, we will describe the structural components of each language and list any extraordinary functional features (for an overview of modern simulation languages, including 'non-CSSL's', with a detailed functional evaluation we refer to Cellier, 1983).

3.1. CSMP and DSL/VS

The simulation language DSL/90 (Syn and Linebarger, 1966) was released by IBM in 1965 and was one of the languages from which CSSL'67 evolved. DSL/90 also served as basis for the IBM products CSMP (IBM 1967), CSMP-III (IBM 1972) and DSL/VS (IBM 1984). Although none of these products adhere to CSSL'67, they all closely resemble the standard.

CSMP is a batch-oriented simulation language using FORTRAN as target language. It features all the recommended CSSL'67 operators and is functionally very similar to the original standard (or vice versa!).

The basic structural elements of CSSL'67 can be found also in CSMP, with the exception of the DYNAMIC section(s) and the segments. CSMP requires the user to include the execution description in the model description file. This precludes any changes in parameter values, required output or run-time control after the translation has been performed and thereby forces the user to either describe the whole execution in advance or iteratively repeat the translation/execution sequence until satisfactory results have been obtained.

There exist several CSMP dialects, the most modern version of which is CSMP-III. Its major improvements over the original CSSL specification are in the area of output representation. CSMP-III includes a rich set of graphics commands including three-dimensional shade and contour plots, cross-plots (trajectories), and over-plots (mergers of variables from several runs). CSMP-III also includes an array integration statement which together with the three-dimensional graphing capabilities allows to use CSMP-III for the solution of simple parabolic and hyperbolic

partial differential equations in one space variable by use of the method-of-lines approach.

DSL/VS is a modernized version of DSL/90 and thereby also very similar to CSMP. It is batch-oriented and uses FORTRAN as target language. In addition to the structural elements of CSMP, DSL supports DERIVATIVE and SAMPLE sections. The SAMPLE section is intended for the modeling of digital controllers in an otherwise continuous system (sampled system), but it can also be used for the modeling of any other isolated discrete action. However, as only one SAMPLE section is allowed, more general combined continuous/discrete simulation is not possible in DSL.

Functionally DSL offers several unique and interesting features:

In addition to the standard integrational step-size and communication interval, DSL supports a sampling rate for the SAMPLE section and separate intervals for numeric and graphical output.

Some algorithms useful in the analysis of systems, like a root-locus algorithm and a discrete FFT, have been included.

Two different optimization algorithms have been included in DSL, making multiparametric, unconstrained optimization possible. Thereby, the program approach to execution control is used, forcing the user to include a small FORTRAN main program.

During the simulation a rudimentary data-manager regulates the flow of numerical results to one or several external files. After the simulation is completed, a graphical postprocessor (GRAFAEL) will be called with which the saved data can be plotted. Apart from normal time histories, trajectories (x versus y) can be obtained and results from different runs can be compared. GRAFAEL supports a range of IBM graphical terminals/plotters and is very flexible in the design and scaling of the plots (automatic or manual scaling, coloring, labeling, tick-marks, line-marks ...). However, as the GRAFAEL commands must be included in the combined model/execution description file, no interactive refinements of the plots can be made.

CSMP and DSL/VS are powerful simulation language for the modeling of pure continuous systems. In addition, DSL enables the simulation of sampled systems and provides the user with a powerful plot-facility. However, both languages are batch-oriented with a combined model and execution description and use a syntax very similar to FORTRAN. They are therefore obsolete compared to systems supporting an interactive experimental frame and free-form input.

3.2. ACSL and CSSL-IV

Both these commercial products very strongly adhere to the CSSL'67 standard, making their basic structures very similar. Both packages are FORTRAN based and run on a large number of different computers ranging from main frames to PC's.

ACSL (Mitchell and Gauthier, Ass., 1981) as well as CSSL-IV (Nilsen 1984) separate the model description from the experimental (execution) description. Whereas the model description is translated and linked in the normal manner, the commands for experimental control are entered interactively, this enables the monitoring and controlling of the simulation(s) using input commands and numerical or graphical output. To illustrate this, we assume that we have written and compiled a

model-description file of a DC-motor with a simple controller and that we want to experimentally calculate the value of a controller-parameter P so that the error function of the motor position is in some way 'optimal'. Suppose we have already translated and compiled the model description. We would then enter the following commands to the run-time monitor:

```
'Interactive run-time commands using the syntax of ACSL'
PREPARE T,ERROR   $  'Save a time-history of the variable error'
                     'for interactive plots'
SET, P = 0·1      $  'Set initial value of parameter P'
START             $  'Start one simulation run'
PLOT T,ERROR      $  'Produce graphical plot of error '
                     'time-history on the terminal'
SET, P = 0·5      $  'Unsatisfactory result of controller,'
                     'try new value for P'
START             $  'Simulate anew'
PLOT T,ERROR      $  'Produce plot of error-behaviour with new P'
...
...
STOP
```

Structurally, both ACSL and CSSL-IV support all CSSL'67 elements except for the segments. The macro facility of the two languages has been extended to form a limited programming language of its own. Using this 'meta-language', models of similar but not necessarily equal sub-systems can be described by one macro.

Apart from the CSSL'67 structures, ACSL lets the user include DISCRETE sections describing actions to be taken at the discrete times during the simulation (discrete events). The scheduling of these events can be made in three ways:

The section contains an INTERVAL statement, in which case the event is executed in fixed intervals (corresponds to SAMPLE in DSL).

The section is executed whenever a dynamic variable crosses a certain boundary (state event). This kind of events enables a numerically safe modeling of systems with discontinuities and variable-structure systems.

The time the section is to be executed is precalculated and entered into a list of future events (time events). Time events are useful to model asynchronous phenomena like start-up and shut-down processes. Also, all simulation languages for discrete systems (e.g. network and queuing modeling) are based upon a time-event scheduler. Therefore, time-events are prerequisites for any combined discrete/continuous simulation language.

With the linear analysis capabilities of ACSL it is possible to study several properties of the simulated system. For example, the eigenvalues of the dynamic system and the steady state of a non-linear system can be calculated. For their flexible use, all analysis commands are incorporated into the interactive experimental frame.

CSSL-IV is probably the simulation language with the most functional elements (over 200). Functions for matrix calculations based upon EISPACK (Smith *et al.* 1974; Garbow *et al.* 1977) and LINPACK (Dungorra *et al.* 1979), linear transfer function operators, FFT's, Bode and Root-Locus plots and a large number of non-linear functions complement the standard integrational operators.

To enhance the user friendliness, the interactive experimental frame of CSSL-IV includes a help-facility for on-line assistance.

3.3. DARE

As in the case of CSMP, also DARE denotes an entire family of software systems rather than one specific simulation language (Korn and Wait, 1978). The best-known language in the DARE family is DARE-P which was the earliest available portable simulation language running on machines as small as PDP-11s. DARE follows the CSSL standard less strictly than the previously described languages. Its major advantages are:

The LOGIC block as a replacement of the INITIAL/DYNAMIC/TERMINAL regions. This concept allows to call the simulation as a subroutine which makes DARE-P easier to connect to a general purpose optimization package. The LOGIC block concept also represents the first step towards separation of model description and experiment description. The model is described in the DERIVATIVE block, whereas the experiment is described in the LOGIC block. Some other DARE dialects (MICRODARE, DESIRE, DESCTOP) even interpret the (BASIC-like) LOGIC block, whereas the DERIVATIVE block is compiled by an ultra-fast compiler, making these software systems 'direct executing' (compilation and linkage of a system of 20 differential equations consumes less than one second).

The graphics post-processor which is decoupled from the simulation language. Results from the simulation are stored in a primitive simulation data base from where they can be extracted by the post-processor. This concept indeed provides for a much extended flexibility than even the generous graphics capabilities of CSMP-III offer. Newer DARE dialects (DARE-INTERACTIVE) offer three-dimensional graphics with hidden lines removed, envelope graphics, split-screen graphics, and colours. Some of the DARE dialects (DARE-INTERACTIVE, DESIRE, DESCTOP) also offer a self-scaling run-time display.

While some of the systems (DARE-P) are batch oriented, others (DARE-INTERACTIVE, DESIRE, DESCTOP) are highly interactive.

DARE-INTERACTIVE also extends the CSSL specification with respect to the type of run-time experiments offered as standard features by including automated sensitivity analysis and automated replication analysis.

3.4. SYSMOD

The SYSMOD language is the newest of those discussed within this paper (Baker and Smart, 1982, 1983, 1984). The language has been developed over the last four years by System Designers Ltd (SDL) on contracts from the Ministry of Defence (MOD) United Kingdom and is intended to replace the CSSL'67 influenced DSL77 language. During the research leading to the definition of the new language, the researchers were involved in the current standardization discussions and reviewed all of the available tools. In particular the language COSY, defined by Cellier, Bongulielmi and Rimvall (1979 and 1981), had substantial influence on the definition of the new language. SYSMOD is very soon to be released for Beta test at an MOD research establishment and plans are being formulated for its launch as a SDL product in the near future.

Structurally, SYSMOD has been heavily influenced by the emerging procedural programming languages Ada and Pascal. It is strongly typed with facilities for user defined simulation types and data structures absent from the existing CSSL's. It

retains, however, the segmentation of the model into the DISCRETE/DYNAMIC/TERMINAL/INITIAL regions with revised semantic interpretation consistent with the complete separation of the model from the experiment.

With SYSMOD, again in contrast to the other CSSL's described in this section, the scope of the descriptive modularization is extended. Models are composed of many SUBMODELS linked to form the composite simulation. Experiments act upon such linked composites and the SUBMODELS and EXPERIMENTS are separately translatable allowing libraries of experiments and standard submodels to be built.

Recognizing the need for more robust simulation environments to be engineered, the SYSMOD team configured a so called 'piecewise continuous' run-time environment for the language. These mechanisms constitute the continuous modeling partitioning of the language and allows the model descriptions with their differential/algebraic equations to structurally change in an entirely general fashion. An example is:

```
IF deflection > 22·0 TOL = 0·001
THEN
  x'' = 4·2*x + 4
ELSE
  SUBMODEL refined_velocity_model(x ← 4);
```

In the above example, a structural change is specified reflecting a need for a switch to a more refined model based upon progress of the deflection variable. The tolerance describes the degree to which the modeler 'cares' about the preciseness of this structural change. To obtain this feature, the implementers had to implant sorting in an entirely general fashion.

SYSMOD is a 'combined language'. It can be used for both DISCRETE and CONTINUOUS systems simulation as well as for combined applications. The DISCRETE region of the model description gives the modeler the flexibility of either event-oriented DISCRETE modeling or combined modeling. Truly discrete models can be formed employing the QUEUE data type, the SCHEDULE, INSERT and REMOVE primitives and a PASCAL like language supporting dynamic memory allocation. The run-time environment is, in the absence of any STATE (continuous) variables, able to progress directly from event to event within the simulation. Combined modeling is supported by allowing access to continuous variables within the DISCRETE region and enabling specifications of state events within the DYNAMIC region employing the WHEN primitive. These facilities extend considerably upon those of the previous CSSL's mentioned in this section.

Emerging from the Ada language efforts has been the notion of the 'environment' in which models are engineered. Consequently, each of the programs which comprise the SYSMOD implementation have been engineered into a routine independent 'environment'. Facilities already exists within the implementation to make the administration of the files generated during SYSMOD generations at all levels an automatic process. For instance editing a MODEL causes removal of all simulation files originating from that model automatically. Other facilities check when building composite models so that interfaces are compatible and build descriptive files giving the user a comprehensive source of information for interaction with the model at run-time etc. The beginnings of the much needed simulation environments are therefore already apparent within the present SYSMOD implementation.

The SYSMOD implementation has so far concentrated on the structural elements needed in any 'complete' simulation environment. Plans exist to include a macro-processor, pre and post simulation processors, advanced experimentation tools, fully extended DISCRETE language etc. When we include these plans, SYSMOD is probably the most comprehensive simulation language on the market today.

4. Elements of future simulation languages

In the last chapter we saw how CSSL languages over the years have been extended with features not included in the original standard. Although many of these features are general enough to warrant their inclusion into any new CSSL standard, most of them are of functional rather than structural/conceptual nature. In this chapter we will elaborate on some of these features and discuss some additional concepts in view of any future standard.

4.1. Combined discrete/continuous simulation

All physical systems are inherently continuous, yet by the simplification/abstraction phase of the modeling we often replace these continuous processes by cruder representations. For example, a diode is often represented by a piecemeal linear function rather than the complicated exponential function (the model contains a discontinuity function). Moreover, the opening of a valve is often modeled as an instantaneous event rather than a fast dynamic process (the model contains a discrete event). Neither of these simplified models can be correctly treated in a pure continuous simulation environment, yet few CSSL languages include the necessary discrete elements (see Oren, 1977).

All CSSL languages support a number of discontinuous functions like relay-functions, limiters and hysteresis functions, yet few CSSL's implement these nonlinear functions in a manner guaranteeing a correct numerical treatment. As all numerical integration algorithms make some kind of polynomial approximation of the time-histories, a pure discontinuity can never be correctly approximated by a straight forward integration (regardless of step-size). Instead, the proper method is to halt the integration at the time the discontinuity is reached, switch to the new section of the non-linear function and restart the integration. Thereby, the exact time of the discontinuity (the state event) must be calculated by iterative interpolation. Several non-CSSL languages like GASP-V (Cellier and Blitz 1976) and SYSMOD support such a correct treatment, in SYSMOD even without the user knowing about it.

Our second illustration, the opening valve (see above), exemplifies the kind of action described in pure time- or state-events. These kind of events were explained in the section on ACSL, which is the only CSSL language supporting these discrete elements. Yet the need for discrete elements in mainly continuous simulations exist, the success of combined non-CSSL languages like GASP-IV (Pritsker, 1974), SLAM (Pritsker and Pegden, 1979) and GPSS-III (Schmidt, 1984a and 1984b) verifies this.

4.2. Experimental frames

Already CSSL'67 supported a certain separation of model description code and execution (run-time) commands. Unfortunately, no clear-cut distinction between the two parts was made and certain language elements are found in the wrong place

(e.g. the integration control was placed in the model description whereas it really belonged to the execution commands). More modern CSSL's, for example ACSL and CSSL-IV, make a distinct difference between the model description file and an interactive experimental frame.

Most standardization approaches, for example the suggestions made within the IMACS/TC3 and SCS/CSSL standardization committees (Crosbie and Cellier, 1980–1984), suggest that the model description should be completely separated from the experiment description. Thereby, the model description is to contain all information needed to describe the physical system (differential equations, discrete sections, physical constants and so on) but should be free of all information related to the experiment (integrational control, required output etc.). Likewise, the experimental description should contain only information related to the experiment. Ideally, these two parts should be totally independent, so that for example a standard experimental description for optimization might be used independently on different models (or vice versa).

In a simulation system supporting a complete separation of the model description from the experimental frame, the commands of the experimental part must be far more flexible than the ones existing in most present-day run-time systems (e.g. in ACSL and CSSL-IV). Here the SYSMOD language and Korn's DESCTOP (1985) both include novel concepts, although they use different approaches:

> The simpler experimental commands of DESCTOP are interactively interpreted with the contention that these experimental commands are 'CPU-frugal' (at least compared to the number-crunching numerical integration), making time lost by the interpretation neglectable.
>
> The highly structured experimental frame of SYSMOD is compiled and thereafter linked together with the model description(s). It can here be argued that a complex experiment description (e.g. an optimization scheme) is far too intricate to be interactively controlled anyway, and that a 'program form' increases the portability and security of each experiment.

The highly interactive and astonishingly flexible user interfaces of recently emerging 'matrix/control environments' like Moler's MATLAB (1980), CTRL-C (Little *et al.* 1984) and IMPACT (Rimvall and Bomholt, 1985) have also initiated a new discussion on the design of experimental frames. Rimvall and Cellier (1985) have shown how a merging of a conventional simulation run-time system with data and program structures of a matrix environment can result in a simple, yet extremely powerful and fully interactive experimental frame.

Yet other suggestions for the structuring of the model/experiments have been given, for example Zeigler's (1981) proposal for three specifications: the model specification, the experimental frame specification and the execution control specification. Thereby, the experimental frame should contain input and output variables, integrational control and run-control (optimization), whereas the execution control would allow for parameter assignments and state initialization. This approach would allow a flexible experimental frame whilst keeping the interactive run-control simple.

4.3. Subprocesses, modularity

As has been shown by Elmqvist (1978), a submodel or macro approach to model structuring where subsystem inputs and outputs have to be predefined is not

modular. A basic example will illustrate this: the simplest electrical component, the resistor, can be modeled as

```
SUBSYSTEM RESISTOR(I = U,R)
  I = U / R
END RESISTOR
```

This definition can then be used for example to calculate the current in parallel connections:

ITOT = *I*1 + *I*2 + RESISTOR(*U*1,500)

On the other hand, the same definition is useless for the calculation of the voltage in a serial connection:

UTOT = *U*1 + *U*2 + ??? RESISTOR(*I*3,*R*) ???

The consequence of this small example is clear: if we want to have *one* model for the resistor, the physical laws only specify one of the two formulae

$I = U/R$ or $U = R \cdot I$

whereas the environment of the submodel indicates which of the formulae should be used. This requires a system where the equations are not only 'vertically' sorted (ordering of complete equations) but also 'horizontally' (ordering within individual equations).

Moreover, a more general notation for the subsystem declaration and interconnection is needed. The syntax of such a general interconnection could be based upon the similar yet independent pioneering work of Elmqvist (1978) and Runge (1977). Both these authors suggest the introduction of general CUTS (interface clusters) over which connections between subsystems are made, either by connecting complete cuts with each other or by 'wire' each variable of a cut separately. The modularity is obtained by two equally important features:

Normally, the directions of the variables declared in a cut (voltage and current in our trivial example) is not specified.

When several systems are connected, variables in the different cuts can be set equal (across variables) or their sum can be set to zero (through variables). Extending our trivial example to three resistors soldered together into a star form. In the connection, the currents would be through variables and the voltages would be across variables:

$I1 + I2 + I3 = 0$

$U1 = U2 = U3$

4.4. Simulation hardware

Due to the number-crunching nature of digital simulations and the impressive size of most general simulation languages, until just a few years ago general simulation programs were found almost solely on larger main frames. However, the advent of modern computer hardware now allows for the implementation of software systems which were unthinkable just a few years ago. In particular, the plunging prices of computer memory and/or new operating systems supporting large address spaces (virtual memory) have permitted the implementation of large simulation packages on smaller machines and the increased processing power of modern pro-

cessors have made such implementations interesting for the simulation of 'real' systems. In a first step, the mini computer (e.g. VAX) brought the simulation tools onto fully interactive, user-friendly machines, taking away some of the black-box character of the main frames. At this time, work stations which combine the flexibility of personal equipment with the number-crunching power of previous main frames are on the market. For example, PC-versions of ACSL and CSSL-IV which run only about 10 times slower than on a VAX are now available. In a few years more powerful workstations with the processing power approximately equivalent to a standard VAX-780 unit will stand on (under) everyones desk. Moreover, such modern workstations (e.g. the APOLLO DOMAIN) offer the user a quite different operating environment; features like windowing, parallel sessions, menu presentation and mouse selections will make the computer accessible to technicians having no formal computer-science education. It is most likely that the simulation environment will follow this general trend. Possible (and very attractive) alternatives to the model specification are described in Elmqvist (1982) and King and Gray (1985), where a graphical editor is used to enter and manipulate the model. This could then be combined with a menu-driven experimental frame for a complete simulation system.

Despite this trend towards personal computing, large size simulations still require number-crunching capabilities far exceeding the processing power of any single CPU. The only solution to this is to use 'more than one CPU', and Karplus (1984) states three possible approaches:

Supercomputers (Cray, CDC 205 etc).

Multiprocessors (Carnegie–Mellon, ETH).

Peripheral array processors (slaves controlled by a 'normal' computer).

Presently, no CSSL language can directly be connected to any of these kind of hardware systems. However, there already exist array processors designed for simulation applications and special purpose multiprocessor systems for ultra-fast three-dimensional operations (three-dimensional run-time display for aircraft simulators etc., three-dimensional real-time data bases) are emerging. Three-dimensional architectures like the INTEL hypercube are to arrive in the next year or two and should manifold the processing power of present workstations (Jefferson *et al.*, 1985a and 1985b). Nevertheless, this is still the playground for hardware manufacturers, so we software engineers have to wait and see. . . .

4.5. Simulation operating systems

Another difficulty with present CSSL's stems from the fact that modern interactive software systems are much more intimately interlinked with the operating software of the executing machine than older batch systems were. Why do editors, for instance, have to be different for every particular piece of hardware? Presently, several independent efforts to cope with this diversity are made:

The plans of the U.S. Department of Defense to replace all computer languages used in their embedded systems with Ada is to be ensued by an Ada environment (KAPSE, APSE). This environment is nothing else but a standardized user-interface to the machine operating systems, with unified editors, file handlers and process controllers.

The spreading of modern workstations supporting windowing techniques might lead to a *de facto* industry standard for the user interface, at least on a superficial level.

A few simulation languages (e.g. some dialects of DARE and TESS from Pritsker, 1984) include their own file-handling and job scheduling. These languages put themselves between the user and the computer operating systems, relieving the user from most operating administration.

Another approach is taken by MIDGET (Rimvall and Cellier, 1984), a 'simulation operating systems'. MIDGET can be seen as a very thick padding between the user and the operating system, giving the simulationist a complete operating environment with a set of menus adequate for the use of a particular simulation language. As these different simulation environments are very similar to each other, the user familiar with one of them (e.g. that of ACSL) should in a few minutes time be able to use the environment of e.g. GASP, SLAM or SDL. MIDGET is presently implemented on a VAX under VMS, in principle however, a system like MIDGET would be implementable on any interactive system, giving the user a machine-independent as well as simulation-system independent interface.

5. Current standardization efforts

During the seventies, the previously described systems emerged as fairly stable pieces of software meeting the needs of a large number of applications. However, we have seen how the advent of modern computer hardware changed the picture completely. Naturally, this called for additional features to be offered. As a consequence, we saw many new simulation languages emerging on the market in the past two or three years which deviate drastically from the CSSL standard. It has become quite difficult for a new simulation user to choose the right tool from the jungle of available systems. For this reason, two standardization groups (IMACS/TC3 and SCS/CSSL) try to come up with a new standard meeting the needs of the eighties and nineties. However, the situation is more difficult today than 1967. Most of the members of both standards committees are 'tool makers' by themselves, and have a large amount of vested interest in having their own system become the new standard. It therefore seems practically impossible to come up with a frozen language which would be accepted as a standard language. Instead, both committees try to come up with a set of features which ought to be included in a modern simulation system. A typical example of such a wish list can be found in Cellier (1983).

The trends in simulation environments are also interesting from a standardization point of view. It is to be expected that all computers (except for the large main frames and super-computers) shall make use of the windowing technique in just a few years from now and that standards like the emerging Ada environment will be in general use. Thus, there is some hope for a *de facto* standardization of the operating software. In the sequel, we may well also encounter a renaissance of the standardization efforts on the language level.

6. Conclusions

Several parallels can be drawn between the simulation standard CSSL'67 and the computer language FORTRAN:

Both can rejoice over an extraordinary long lifetime in the fastest moving field: the computer industry.

Neither product would be alive, if it has not been revitalized over the years (e.g. FORTRAN 77 and ACSL/CSSL-IV).

Both products are momentarily plagued by an abundance of dialects. Each of these dialects has its virtues and the temptation to leave the standard is often irresistible.

Presently, standardization efforts are undertaken to prepare new standards for the eighties. A big problem is that everybody wants to see *his* dialect as new standard.

Competitive products are arriving from all sides to threaten the standards (e.g. Ada, Pascal and SYSMOD, DESCTOP).

Will these two standards survive? Unfortunately, FORTRAN has a chance, whereas the *far better* CSSL standard has less chance of survival. However, the strict CSSL standard will most probably live on in the form of a list of features to be included in future simulation languages. We can therefore expect several systems using different dialects, but with comparable (and quite impressive) capabilities.

7. Acknowledgements

We would like warmly to thank Mr. N. Baker, SDL, United Kingdom, for giving us the latest information on the SYSMOD language. We also express our gratitude to Mr. E. Thaler and the KOMETH team, ETH, Switzerland, for whom no computers are too far away from one another to form a communication link.

References

AUGUSTIN, D. C., STRAUSS, J. C., FINEBERG, M. S., JOHNSON, B. B., LINEBARGER, R. N., and SANSOM, F. J. (1967). The SCi continuous system simulation language (CSSL). *Simulation*, **9**, 281–303.

BAKER, N. J. C., and SMART, P. J. (1982). The SYSMOD language and run time facilities definition, Techn. Note 6.82, Royal Aircraft Establishment, Farnborough, Hampshire, United Kingdom.

BAKER, N. J. C., and SMART, P. J. (1983). The SYSMOD simulation language. In W. Ameling (Ed.), *Proceedings ESC'83*. (Springer Verlag, Berlin) 12723–2.

BAKER, N. J. C. and SMART, P. J. (1984). SYSMOD—An environment for modular simulation. *Proc. SCSC 1984*. In W. Wade (Ed.). (North Holland, Amsterdam).

CELLIER, F. E., and BLITZ, A. E. (1976). GASP-V: a universal simulation package, In Dekker, L. (Ed.), *Simulation of Systems, Proc. of the 8th AICA Congress*. (North-Holland, Amsterdam).

CELLIER, F. E., and BONGULIELMI, A. P. (1979). The COSY simulation language. In Dekker, L., Savastano, G. and VanSteenkiste, G. C. (Eds.), *Simulation of Systems, Proc. of the 9th IMACS Congress*. (North-Holland, Amsterdam).

CELLIER, F. E., RIMVALL, M. C., and BONGULIELMI, A. P. (1981). Discrete Processes in COSY. In Maceri, F. (Ed.), *Proc. of the European Simulation Meeting held in Cosenza, Italy. (April 1981)*. Also in Crosbie, R. E. and Cellier, F. E. (Eds.), TC3-IMACS, *Simulation Software, Committee Newsletter*, No. 11. (July 1982).

CELLIER, F. E. (1983). Simulation software—today and tomorrow. In Burger, J. and Jarng, Y. (Eds.). *Proceedings IMACS-Conference, Nantes, France*. (North Holland, Amsterdam).

CLANCY, J. J. and FINEBERG, M. F. (1965). Digital simulation languages, a critique and a guide. *AFIPS Conference Proceedings Vol. 27*. Spartan Books, Washington D.C.

CROSBIE, R. E., and CELLIER, F. E. (Eds.) (1980–84). Committee Newsletter on Simulation Software no. 6–12. Technical Committee 3 (simulation software) of IMACS.

DUNGORRA, J. J., BUNCH, J. R., MOLER, C. B., and STEWART, G. W. (1979). *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics.

ELMQVIST, H. (1975). SIMNON—An interactive simulation program for nonlinear systems—user's manual, Report TFRT-3091, Dept. of Automatic Control, Lund Institute of Technology, Lund, Sweden.

ELMQVIST, H. (1977). SIMNON—An interactive simulation program for nonlinear systems, in Hamza, M. H. (Ed.), *Proc. of the International Symposium SIMULATION'77*. (Acta Press, Anaheim, Calgary and Zurich).

ELMQVIST, H. (1978). A structured model language for large continuous systems. PhD Thesis, Dept. of Automatic Control, Lund Institute of Technology, Report: CODEN: LUTFD2/(TFRT-1015), 226 p.

ELMQVIST, H. (1982). A graphical approach to documentation and implementation of control systems. *Proc. 3rd IFAC/IFIP Symposium on Software for Computer Control*, SOCOCO82. Madrid, Spain.

GARBOW, B. S., BOYLE, J. M., DONGARRA, J. J., MOLER, C. M. (1977). *Matrix Eigensystem Routines, EISPACK Guide Extensions*. Springer, Lecture Notes in Computer Science, *51*.

GRANDA, J. J. (1983). Computer Aided Modeling Program (CAMP), a bond graph preprocessor for computer aided design and simulation of physical systems using digital simulation languages. Dissertation, University of California, Davis.

HARNETT, R. T., SANSOM, F. J., and WARSHAWSKY, L. M. (1964). MMIDAS, an analog approach to digital computation. *Simulation*, **3,** (1964), 3.

IBM (1967, 1972). *System/360 Continuous System Modeling Program. User's Manual*, Program Number 360A-CX-16X, Form GH20-0367-4.

IBM (1972). *Continuous System Modeling Program III (CSMP III) Program Reference Manual*, Program Number 5734-XS9, Form SH19-7001-2, IBM Canada Ltd., Program Product Center, 1150 Eglington Ave. East, Don Mills 402, Ontario.

IBM (1984). *Dynamic Simulation Language/VS (DSL/VS). Language Reference Manual*, Program Number 5798-PXJ, Form SH20-6288-0, IBM Corporation, Dept. G12/Bldg. 141, 5600 Cottle Road, San Jose, CA 95193.

JEFFERSON, D., and SOWIZRAL, H. (1985a). Fast concurrent simulation using the time warp mechanism. In *Proc. Distributed Simulation, 24–26 January 1985, San Diego, California* (SCI-publications, La Jolla, Calif.)

JEFFERSON, D., BECKMAN, B., HUGHES, D., LEVY, E., LITWIN, T., SPAGNUOLO, J., VAVRUS, J., WIELAND, F., and ZIMMERMAN, B. (1985b). Implementation of time-warp on the CALTECH hypercube. In *Proc. Distributed Simulation, 24–26 January 1985, San Diego, California* (SCI-publications, La Jolla, Calif.)

KARPLUS, W. J. (1984). Selection criteria and performance evaluation methods for peripheral array processors. *Simulation*, **43,** 125–131.

KING, R. A., and GRAY, J. O. (1985). A flexible data interpreter for Computer Aided Design & simulation of dynamic systems. In *Proc. 3rd IFAC Symposium on Computer Aided Design in Control and Engineering Systems*. Copenhagen, July 31–August 2, 1985. (Pergamon Press, Oxford).

KORN, G. A. (1985). *DESCTOP Reference Manual, Version V2.0*. University of Arizona, Tucson, AZ 85721.

KORN, G. A., and WAIT, J. V. (1978). *Digital continuous-system simulation*. (Prentice Hall, Englewood Cliffs, N.J.) 212 p.

LITTLE, J. N., EMANI-NQEINI, A., and BANGERT, S. N. (1984). CTRL-C and matrix environments for the computer aided design of control systems, *In Proc. 6th International Conference on Analysis and Optimization (INRIA)*, (Lecture notes in Control and Information Sciences 63, Springer Verlag).

MITCHELL and GAUTHIER, ASSOC. (1981). *ACSL: Advanced continuous simulation language—User Guide/Reference Manual*. P.O. Box 685, Concord, Mass.

MOLER, C. (1980). *MATLAB, User's Guide*. Department of Computer Science, University of New Mexico, Albuquerque, USA.

NILSEN, R. N. (1984). *The CSSL-IV Simulation Language, Reference Manual.* Simulation Services, 20926 Germain Street, Chatsworth, California.

OREN, T. I. (1977). Software for simulation of combined continuous and discrete systems: A state-of-the-art review. *Simulation*, **28**, 33–45.

PETERSON, H. E., and SANSOM, F. J. (1965). *MIMIC—A digital simulation program.* SESCA Internal Memo 65-12, Wright Patterson AFB.

PRITSKER, A. A. B. (1974). *The GASP IV simulation language.* (Wiley, New York).

PRITSKER, A. A. B., and PEGDEN, C. D. (1979). *Introduction to simulation and SLAM.* (Halsted Press, New York and Systems Publishing Corp., West Lafayette).

PRITSKER & ASSOCIATES (1984). *The TESS User's Manual.* P.O. Box 2413, West Lafayette, IN 47906, USA. 515 p.

RIMVALL, M., and CELLIER, F. E. (1984). MIDGET, Ein flexibles, simulationstechnisches Entwicklungssystem. In Breitenecker, F., and Kleinert, W. (Eds.) *Proc. of the ASIM'84 Symposium, Vienna, Austria, September 25–27, 1984.* Springer, Informatik Fachberichte.

RIMVALL, M., and BOMHOLT, L. (1985). A flexible man–machine interface for CACSD applications. *In Proc. 3rd IFAC symposium on Computer Aided Design in Control and Engineering Systems. Copenhagen, July 31–August 2, 1985.* (Pergamon Press, Oxford).

RIMVALL, M. and CELLIER, F. (1985). The matrix environment as enhancement to modeling and simulation. *In Proc. 11th IMACS world conference, Oslo, Norway. August 5–9, 1985.*

RUNGE, T. F. (1977). A Universal Language for Continuous Network Simulation. PhD Thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Report: UIUCDCS-R-77-866, 153p.

SCHMIDT, B. (1984b). Der Simulator GPSS-FORTRAN Version 3. Springer Verlag, *Fachberichte Simulationstechnik*, **2**.

SCHMIDT, B. (1984b). Modellbilding mit GPSS-FORTRAN Version 3. Springer Verlag, *Fachberichte Simulationstechnik*, **3**.

SMITH, B. T. *et al.* (1974). Matrix Eigensystem Routines, EISPACK Guide. Springer, *Lecture Notes in Computer Science*, **6**.

SYN, W. M., and LINEBARGER, R. N. (1966). DSL/90—A digital simulation program for continuous system modeling. *AFIPS Conference Proceedings, 28.* Spartan Books, Washington, D.C.

ZEIGLER, B. (1981). A Methodology for simulation Program Development. In Crosbie, R. E. and Cellier, F. E. (Eds.), *TC3-IMACS, Simulation Software, Committee Newsletter, No 10.* (September 1981).