MUSS, A MODULAR SIMULATION SYSTEM

A. Guasch *
Institut de Cibernetica
Polytechnic University of Catalonia
Barcelona - SPAIN

ABSTRACT

In contrast with classical simulation languages, the present trends are evolving towards fully integrated interactive modelling and simulation environments. These environments have to combine interdisciplinary techniques such as expert systems, object oriented programming and data base management.

To achieve the above objectives, the architecture of the simulation programming language and that of the run-time simulation environment which exercises the models should be designed allowing modularity and flexibility. Furthermore, the robustness of the environment should be reinforced.

In this paper the *MUSS* simulation system is presented, emphasizing the innovative concepts: the hierarchical architecture of the *MUSS* simulation language, the preprocessor analysis and segmentation phases and the structure of the run-time simulation environment.

INTRODUCTION

The research lines of the *Institut de Cibernetica (IC)* are strongly influenced by the demands of the industrial and scientific communities in Catalonia. Among them, those areas mainly related with simulation are Automatic Control (Electrical Engineering) and Bioengineering.

In the seventies, simulation in the *IC* was based on hybrid techniques (Huber et al. 1982). Thereafter, the simulation group started to work in pure digital simulation languages. As a result, the *ICDSL* CSSL-like simulation language was designed (Guasch et al. 1984).

About 1983, the lack of modularity of the *ICDSL* language was noticed as a significant constraint in the simulation of large systems. Therefore, the design and implementation of the *Modular Simulation System (MUSS)* got under way.

A hierarchical architecture sustaining a modelling, coding and testing bottom-up approach has been chosen and the research guide line has been to provide the theoretical and practical background needed to support a modular structure without restricting the following general objectives:

- The language should be declarative and should manage a modularity coherent with the division of the physical system into subsystems through a minimal but sufficient number of different blocks.
- The separation between the model description and the experiment should be done in such a way that the model remains unchanged along the experiments and ready to be used from another model (submodel) at the end of the validation and verification experiences.
- It should be possible to build a system model in a bottom-up way relating two or more submodels from a model or submodel of higher hierarchical level.
- Isolated preprocessing of submodels as well as run time symbolic access to all the variables should be able.
- The language should be designed in order to be easily extendible to real time applications.

The *MUSS* architecture proposed joins the object oriented language concept and has been conceived having in mind to support in a future concurrent programming and AI reasoning.

MUSS ARCHITECTURE

The architecture of the MUSS language and related language constructs converge with the trends on piecewise-continuous system simulation languages (Crosbie and Hay 1982; Hay et al. 1985a) and with the state of the art on combined simulation languages (Smart and Baker 1984; Oren 1984; Kettenis 1986).

Although the *MUSS* language has been designed to initially support the continuous time modelling formalism, the simulation environment has been conceived to easily expand the language to combined models. As an example, the *class concept* which allows the generic instantiation of processes is supported although at present, only continuous processes are handled.

Simulation program

A *MUSS* program is composed of a set of blocks whose structure guarantees the proposed objectives Three types of blocks may be present in a program:

- Submodels: provide the user with mechanisms to describe a physical subsystem. A submodel block may call and may be called by none, one or more submodel blocks. A *MUSS* simulation model is composed by a set of submodels. A model is a relative concept which depends on the experiment block being executed.
- Experiments: control the execution of a single evolution. None, one or more models can be called from the dynamic region of an experiment. In the experiment block, mechanisms for performing a set of evolutions -multi-run study- are not provided. Experiments can not call one each other.
- Studies: A study block controls the execution of a set of evolutions -experiments- One or more experiments can be invoked from the dynamic region of the study.

A program alone does not have necessarily to define a complete environment, neither a study or experiment ready to be executed, neither a model. The set of preprocessed and compiled blocks belonging to a program are put together in a chosen library. Later on, a given environment will be set up by an environment generator which selects study or/and experiment blocks from object libraries

Submodels

Most of the commercial currently available continuous system simulation languages are based on the SCi CSSL report (Strauss 1967). In its implementation the most important part is concerned

----

* At present, he is a visiting research professor at California State University, Chico thanks to the support of the Fulbright institution and of the Spanish Ministry of Sciences and Education.

with numerical algorithms while the programming structures are relatively poor (Brennan 1968; Chu 1969). In these languages, the only way to achieve modularity consists on the use of MACRO pseudoblocks or preprocessor target language subroutines (most often, FORTRAN subroutines). Although the use of MACRO blocks as a basic element to achieve model modularity still have adepts (Nilsen 1982; Breitenecker 1983), the general feeling is that MACROS are still needed because independent translation of submodel code is not always possible (Cellier 1979; Baker and Smart 1983; Freeman and Benyon 1984; Korn 1987).

In contrast, new simulation languages offer a higher level of modularity in, at least, two aspects :

- They allow program blocks consistent with the division of the physical system into subsystems.
- They allow the automatic building of the model by the use of submodels.

Oren (Oren 1979b) defined the concept of *modular coupled model*. A modular coupled model consist of several submodels where coupling specifications define the input/output relationships between submodels. Depending on the disposition of each submodel relative to other submodels, two types of modular coupled models can be defined:

- *Hierarchical model*: several coupled submodels where at least one of the submodels is itself a coupled model.
- *Flat model*: several coupled submodels. The submodels are not themselves coupled models.

MUSS submodel block

MUSS uses the hierarchical modelling approach. Submodel blocks may be translated in isolation which helps to make the modelling turnaround time shorter than languages that need to translate all the submodel and experiment blocks each time a change is made on a submodel or experiment block.

The submodel block consists of the classical two regions, an *initial region* and a *dynamic region* plus a *static region* which is equivalent to the *static structure* of *GEST* (Oren 1984). The static region is described basically in terms of model descriptive variables such as state, input and output variables and constants and parameters.

Although the inclusion of the classical *terminal region* has been rejected because the calculation to be performed when a finish conditions meet can always be included in the experiment, its incorporation in the submodel block can be done without restricting the proposed objectives.

MUSS experiment block

A simulation experiment is defined as *a simulation run over a period of time from a known initial frame* (Symons 1986). Unlike currently developed simulation languages, the *MUSS* experiment block monitors the execution of a single simulation run in contrast with the other languages whose experiment descriptions may monitor the execution of a set of runs.

We rather distinguish between a simulation experiment and a simulation study. The study block described in the next section provides the mechanisms to perform a set of related experiments.

A model without an experiment can not be executed. Even though an experiment may be called by a study. The experiment can always be optionally executed in independence with respect to the study.

In the most general case an experiment block may have three segments: a dynamic segment, a control segment and an output segment. The dynamic segment has the classic three regions: initial, dynamic and terminal regions plus a static region.

MUSS study block

The study block monitors the execution of a set of experiments -simulation study-. Usually, the study block will be called from the *MUSS* simulation environment. The study block may be optionally called from sophisticated main programs coded by the users. Moreover, the study itself may be supplied by the user in C target code, which in turn, calls the experiment blocks through a clear set of interfacing routines.

Like the experiment block, in the most general case a study block may have three segments: a dynamic segment, a control segment and an output segment. The dynamic segment has four regions: initial, dynamic, terminal and static regions.

SUBMODEL ANALYSIS

Monolithic simulation languages are not flexible enough to easy the task of studying models of high complexity, even for expert users. New structures -macro, sample, submodel, module, model- have been added to simulation languages to increase its modularity, but this effort to increase the modularity has not been extended to increment the 'intelligence' of preprocessors and compilers for simulation languages.

Different aspects regarding the robustness of simulation software have been described in (Elzas 1979) an expanded later in (Cellier 1984). In the design of the MUSS system and special care has been given to its robustness:

- The hierarchical structure of the *MUSS* simulation language is suitable for the division of the real system into subsystems.
- Redundancy is introduced in the submodel code. For example, the user is forced to declare all the submodel variables.
- The use of LALR(1) grammars to specify the MUSS language increases the robustness of the *MUSS* preprocessor with respect to its maintainability.
- The *MUSS* preprocessor ensures that syntactical errors will not propagate from the preprocessor to the *C* compiler stage.
- The *MUSS* preprocessor performs extensive error checking looking for model consistency and completeness.
- *MUSS* relies on reputed numerical algorithms increasing the robustness of the run-time system (*MUSS* simulation environment).

In the analysis of the submodel code four functional main phases can be distinguished (Guasch 1987):

- *Code consistency checking*: This stage, looks at the code consistency.
- *Submodel dynamic initialization analysis*: In this phase, besides checking that the submodel can be properly initialized, the executable code needed for initializing the current submodel (this includes the discontinuous functions initializations) and the called lower level submodels is grouped into the *initial segment*.
- *Discontinuous function computations analysis*: During this phase, the code needed to evaluate the discontinuous functions of the current submodel and those in the called lower level submodels is grouped into the *discontinuous segment*. Moreover, discontinuous functions are classified in order to generate run code reducing the time overhead at event occurrences.
- *Dynamic computations*: During this phase, the code needed to calculate derivatives is grouped into the *ODE segment* (Ordinary Differential Equations segment).

Submodel sorting

As Clancy (Clancy and Fineberg 1965) states, the development of a sorting method by Stein (Stein and Rose 1960) was an important step towards the design of more powerful and flexible Continuous System Simulation Languages (CSSL). Since then, this feature has been provided by many widely used CSSL languages.

The automatic sorting of the sentences makes free the user from the responsibility of ensuring a proper execution order of the simulation model code. This important feature should be supported, in our opinion, by modern simulation languages.

The *MUSS* language has been conceived as declarative and its architecture hierarchical. The sorting algorithm which has to convert the source code into a procedural one faces a problem not found in classical monolithic architectures, that of the *information loops*.

A sentence in the dynamic code in which a submodel is invoked is formally equivalent to an assignment statement: it has a set of input and output variables (the submodel interface). The difference arises from the fact that the coupling of the variables in the interface through the called submodel code is hidden to the preprocessor sorting procedure. If that procedure detects algebraic loops involving interface variables, the loop may be really algebraic -which would be the case if the above mentioned coupling is algebraic- or merely an *information loop*.

Known approaches to avoid information loops are:

- Handle the submodels as MACRO's. The statements of the called submodels will be spread over the statements of the calling submodel. In this case, all the submodels must be able to be retrieved in source form. Moreover, MACRO-like facilities should be provided. Furthermore, the time spent at preprocessing time increases because of the necessity of translating all the lower level submodels which have to be handled like MACRO's.
- Force either the submodel input variables or the submodel output variables to be all state type. This approach, in our opinion, is restrictive because the correspondence between the physical subsystem and the submodels in the hierarchical model can be lost.
- Force the user to separate the computation of the derivatives from the output computations which are assembled in a specific block in which the outputs only depend on state variables. The main objection to this approach is that the user is forced to bother about requirement imposed by restrictions in sorting capabilities. Moreover, a different type of submodel has to be defined for coding subsystems when the submodel output variables are algebraically related to the submodel input variables.

The method used by the *MUSS* preprocessor (Guasch 1987), based on the segmentation of the ODE segment, does not impose restrictions on the submodel architecture neither in the hierarchy. It is based on the segmentation of the ODE segment into subsegments (*state, algebraic* and *derivative* segments) which can be characterized by the following structural properties:

- *State segment*: submodel output variables in it depend only on parameters, constants or state variables. Therefore, they may not exist pure algebraic chains between input and output variables.
- *Algebraic segment*: it clusters the input-output algebraic computations.
- *Derivative segment*: computations involving output variables are not allowed. Derivative computations, *when* clauses and computations to be performed at communication intervals will be assembled in this segment.

Therefore, at preprocessing time, any call to a lower level submodel will be splitted into several calls, one for each submodel segment (initial, discontinuous, state, algebraic and derivative segments).

RUN-TIME SIMULATION ENVIRONMENT

Following the generally accepted software engineering principles proposed by (Oren and Zeigler 1979a), the experimentation with models has to be completely separated from models themselves. The architecture of *MUSS* goes one step forward separating experiments from studies increasing the modularity of the simulation environment. The concept of modularity is one of the most important concepts of structured programming (Golden 1985). Nevertheless, modularity alone is not enough to produce well designed programs but it helps to increase the reliability of simulation software.

In this section, we will analyze the structure of the *MUSS* simulation environment. Figure 1 represents models, experiments and studies in an user defined interactive simulation environment

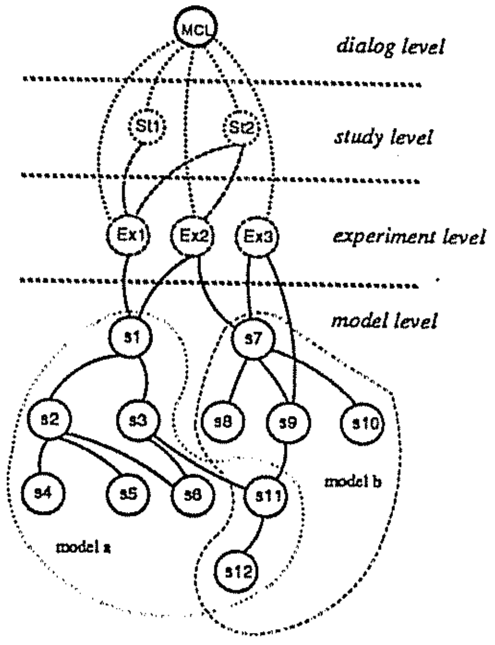

Figure 1

The models appear at the lowest level of the hierarchy. They are composed by a hierarchical set of submodel blocks. From the user perspective a model is a tree of submodels (i.e. submodel *s11* called from submodel *s3* does not model the same physical subsystem as submodel *s11* called from from submodel *s9*). From the implementation point of view, the submodel hierarchy can better be handled as a digraph.

In the next bottom-up level of the hierarchy, experiment blocks appear. Experiment blocks may call zero, one or more models. Its goal is to control a run.

The next-up level may include study blocks. Its objective is the control of model experimentation (i.e. optimization, identification, sensitivity analysis).

The *dialog level* is on top. In the dialog level the *MCL* (MUSS Command Language) language is used to communicate with the simulation environment. An user defined simulation environment may include a large number of models, experiments, studies and data files. Thus, a good management of the environment is very important. To achieve that goal, the MCL has been designed in order to provide the users with a friendly interface with the *MUSS* environment. MCL can be seen as the monitor of the simulation environment. Through it, information about any lower level block can be got. Moreover, From the dialog level experiment or study instances of any experiment or study present in the environment can be created and activated for execution.

Model structure

The structure of *MUSS* models has been designed to solve the main problems related to the hierarchical modelling approach and the separate compilation of submodels of the *MUSS* simulation system. These problems are:

- *Reentrance*: a private data storage area must be allocated and handled for each submodel instance.
- *Symbolic access*: The present *MUSS* prototype allows symbolic access to all submodel variables and access to the submodel information stored in a model data base.
- *Dynamic memory management:* each piecewise continuous model can also be seen as a *generic model* but does not actually occupy data storage. An instance may be created, called and destroyed implicitly. To create and destroy a model instance implies allocation and deallocation of data storage private to each submodel instance.
- *Model initialization*: the static initialization is performed when instances of continuous processes are created and the dynamic initialization is performed before each each simulation run (experiment) when the initial segments are executed.

MUSS command language (MCL)

The Muss Command Language (MCL) is the language through which simulation users communicate with the *MUSS* simulation environment. MCL contains an extensive friendly set of commands that allows users to do tasks such us:

- Get information about models, experiments and studies present in the simulation environment.
- Execute selected studies and/or experiments.
- Edit and execute MCL command files.
- Get run time statistics from instrumental variables in the system.

Some of the most representative commands are:

- *show*: displays information about the user defined simulation environment.
- *create*: creates active versions (instances) of studies and experiments present in the environment.
- *set block*: sets the default block (study, experiment or submodel block). The default block can be directly accessed.
- *type variable*: displays the values of study, experiment or submodel variables, parameters or constants.
- *do*: invokes for execution an active version of a study or an experiment.
- *remove*: delete an active version of a study or an experiment.

PRESENT IMPLEMENTATION STATE

A prototype of the simulation environment has been completed and successfully tested. Its core embraces the following main modules:

- The MCL interpreter, it has the responsibility for understanding users' commands. It has been coded using automatic compiler production techniques.
- The *MCL executive* which embodies the set of algorithms achieving the users' commands. It includes those processes involved in study, experiment and model instantiation, activation, execution and deletion. It also allocates the dynamic memory necessary for the integration package.
- The *LSODAR integration and root finder* package. It is written in Fortran. Some minor changes have been introduced to properly interface it with the *MUSS* environment.
- The *LSODAR front-end* routine. Its main tasks are: set the base addresses of the working memory areas; set set the discontinuous states associated to the discontinuous functions at initial time and at each event occurrence; invoke I/O tasks at each communication point; and call the LSODAR package with the proper input arguments.

A prototype of the *MUSS* preprocessor is still being coded.

CONCLUSIONS

The main results derived from this research project are:

- The proposed architecture of the *MUSS* language is coherent with the natural division of the physical system into subsystems trough the minimal but sufficient number of blocks which have been defined.

- The *segmentation concept* contributes to the reliability of the simulator software. Splitting up the submodel code into the initial, ODE and discontinuous segments is consistent with the functional tasks involved in a simulation run.

- *Isolated preprocessing* of the submodel, experiment and study blocks is allowed. It has been achieved, avoiding restrictions, by the division of the ODE segment into the state, algebraic and derivative subsegments.

- The design of the *MUSS* strengths a modularity converging to the *object oriented* language concept. The *MUSS* run-time simulation environment sefinition is close to that of object oriented languages and therefore it can be considered a good starting point to embody reasoning; on the other hand, its architecture which differentiates four levels (dialog, study, experiment and model) opens the door to introduce *Artificial Intelligence (AI)* techniques to each one independently.

Recent studies suggest that object-oriented methodologies and object-oriented programming languages can contribute to the design of more powerful simulation environments.

REFERENCES

Baker N.J. and P.J Smart. 1983. "The SYSMOD simulation language". *The European Simulation Congress ESC 83.* (Sept.): 281-286.

Breitenecker F. 1983 "The Concept of Supermacros in Today's and Future Simulation Languages". *Math. and Comp. in Sim* (June): 279-289.

Brennan R. D. 1968. "Continuous Systems Modelling Programs : State-of-the Art and Prospects for Development". *Proc. of the IFIP Working Conference on Simulation Programming Languages.*

Cellier F. E. 1979."Combined continuous-discrete system simulation languages.. usefulness, experiences and future development". *Methodology in systems modelling and simulation*. Ed. Zeigler et al. North Holland: 201-220.

Cellier F. E. 1984. "How to Enhance the Robustness of Simulation Software". *Simulation and Model-Based Methodologies: An Integrative View.* Ed. "Oren T. I. : 519-536.

Chu. 1969. *Digital Simulation of Continuous Systems.* McGraw-Hill Book C.

Clancy J. J. and M.F. Fineberg. 1965. "Digital Simulation Languages, a critique and a guide". *AFIPS Conference Proceedings.* Vol. 27

Crosbie R. E. and J.L. Hay. 1982. "Towards new standards for continuous-system simulation languages". *Proceedings of the 1982 summer Computer Simulation Conference.* (July): 186-190.

Elzas M. S. 1979 "What is needed for robust simulation ?". *Methodology in Systems Modelling and Simulation.* ed. Zeigler et al., North-Holland.

Freeman T. G. and Benyon P. R. 1984 "Comments on the proposal for a new simulation language standard". *TC3-IMACS Simulation Software Committee Newsletter*, no. 12 (Aug.).

Golden D. G. 1985. "Software engineering considerations for the design of simulation languages". *Simulation* (4), (Oct.): 169-178.

Guasch A., Huber R. and Ilari J. 1984. "ICDSL (Instituto de Cibernetica Digital Simulation Language)."*IFAC Simposium, Automatica en la Industria* Zaragoza. (Nov.): 349-356

Guasch A. 1987. *MUSS: A contribution to the structural analysis of continuous system simulation languages.* Ph.D thesis. Universitat Politecnica de Catalunya. Barcelona.

Hay J. L., Pearce J. G.,Turnbull L. and Crosbie R. E. 1985 *ESL software user manual* (April).

Huber R. M., Basanez L.,Juan R. A. and Fernandez R. O. 1982 "Hybrid computation experience at the Instituto de Cibernetica, Spain." *Proc. 10th IMACS World Congress on Syst. Simul. and Scient. Comp.* Montreal. Vol.4. (Aug.).

Kettenis D. L. 1986. "Cosmos: A Member of a New Generation of Simulation Languages." *Proc. of the 2nd European Simulation Congress.* Antwerp, Belgium. (Sept.): 263-269.

Korn G. A. 1987. "A new software technique for sub-model invocation". *Simulation.* Vol. 48 (3). (March): 93-97

Nilsen R. N. 1982. "Macros Make More Meaningful Models". *Proceedings of the 1982 summer Computer Simulation Conference.* Denver. (July): 17-23

Oren T. I. and Zeigler B. P. 1979a. "Concepts for Advanced Simulation Methodologies". *Simulation.* Vol. 34 (3). (March): 49-82.

Oren T. I. 1979b. "Concepts for Advanced Computer Assisted Modelling." *Methodology in Systems Modelling and Simulation.*: 29-55.

Oren T. I. 1984. "GEST - A modelling and simulation language based on system theoretic concepts." *Simulation and Model-Based Methodologies: An Integrative View.*: 281-335.

Smart P. J. and Baker N. J. C. 1984. "SYSMOD - An environment for modular simulation." *Proc. 1984 Summer Computer Simulation Conference.* Boston. (July): 77-82.

Stein M. L. and Rose J. 1960. "Changing from Analog to Digital Programming by Digital Techniques." *JACM.* Vol. 7: 10-23.

Strauss J. C. 1967. "The SCi Continuous-System simulation language". *Simulation.* Vol. 9 (6). (Dec.): 281-303.

Symons A. 1986. "Summary of some current issues". *TC3-IMACS Simulation Software Committee Newsletter.* No. 86/01. (Jan.).