CONTINUOUS SIMULATION IN SMALLTALK

Antoni GUASCH

Departament d'Enginyeria de Sistemes
Universitat Politecnica de Catalunya
Diagonal 647 - 2 planta
08028 Barcelona
guasch@esaii.upc.es

Department of Computer Science
California State University, Chico
Chico, CA 95929-0410

ABSTRACT

Although object-oriented languages such as Smalltalk do not seem very suitable for continuous simulation, they offer a set of characteristics and programming environments that are excellent for prototyping.

This paper describes the previous author's experience developing continuous model simulation environments using FORTRAN and C programming languages and the current results obtained using Smalltalk. The transition from FORTRAN to C, from C to C++ and later on, the move to Smalltalk has been propelled by the need to develop more flexible and easy to use simulation environments.

A continuous modeling and simulation environment has been implemented in the Smalltalk\V 286 system. The simulation environment, called SIMBIOS, strongly supports hierarchical modeling and simulation. In this system, users can examine their models in a highly flexible environment. Since interactive simulation is supported, the usual sequence of editing, translating, compiling, linking and simulating the model is avoided.

INTRODUCTION

From 1980 to 1983 a CSSL-like continuous system simulation language was developed and coded using FORTRAN 77 (Guasch 1984). This language has been and is still used for research, industrial and academic projects. At that time, the limitation of the CSSL67 based simulation languages were well known: syntactic deficiencies, hierarchical model were not allowed and the discontinuity handling mechanisms were incomplete (Oren 1975). Therefore, a basic research project was undertaken to solve such problems; as a result, a new approach to hierarchical modeling was proposed (Huber 1986).

To test these ideas, a simulation environment prototype was developed from 1986 to 1987. This prototype called MUSS ("ModUlar Simulation System") was coded in C (Guasch 1987). The move from FORTRAN to C was done in order to support hierarchical structures and submodel instantiation. Moreover, the availability of Lex and YACC lexical and syntactical tools was also an important aspect. However, the C prototype could not be completed because its complete development required important manpower.

Fortunately, the most important aspects of the proposed hierarchical and modeling approach can be easily implemented using object-oriented programming languages. Therefore, a first simple object-oriented simulation prototype was coded in C++ (Guasch 1989). However, a later move to Smalltalk was done because the Smalltalk software development environment was found to be much more suitable than C++ for fast prototyping.

Since 1989, we have been developing a Smalltalk-based continuous simulation environment in collaboration with the Departament of Computer Science and the McLeod Institute of Simulation Sciences at California State University, Chico (Guasch 1990). The system is called SIMBIOS ("SIMulation Based on Icons and ObjectS").

The present SIMBIOS prototype includes over 200 classes and several thousand methods. These classes make up a rich simulation environment that includes, among other tools, an icon modeling interface; a powerful scope window; submodel specific windows; integration, state and time event handling algorithms; and hierarchical model support.

The benefits of using Smalltalk are mostly derived from its intrinsic object-oriented methodology. However, it has been found that our programming productivity is considerable higher than in C++ thanks to the dynamic binding capabilities and powerful development tools of the Smalltalk environment (Dyke 1989).

OBJECT ORIENTED PROGRAMMING (OOP)

Object-oriented methodologies and programming languages attributes have been extensively discussed in the scientific literature (Cox 1986; Meyer 1988; Weiner 1988). It is not our intention to gain a deep insight on the topic but to offer our particular experience in the area.

Object-Oriented Programming languages support the following properties (Pinson 1988): data abstraction, encapsulation, inheritance and polymorphism. Each one of these properties can contribute to increase the flexibility and modularity of the simulation environment. With software maintenance accounting for a very high proportion (Meyer 1988) of software cost, this flexibility is very important. It manifests itself in two ways in particular: extensibility, the capacity to be extended and altered throughout the software life cycle, and reusability, a measure of the extent to which code from one application can be utilized in another.

Our two year experience working with object-oriented programming languages confirms the previous statement:

- Several aspects of the SIMBIOS prototype have been developed by students as part of their Master's project at the Universitat Politecnica de Catalunya, Barcelona and California State University, Chico. The used methodology eases the task of integrating the students code with the prototype. Furthermore, it is now possible to check it in a shorter period of time. This is possible because the stress has shifted to the structure of the project in classes and methods rather than on the structure and code of each subroutine.

- The existing code can be easily reused by different applications. For example, we have two students developing a control package prototype. They did not have to code a window for the representation of variables in the time domain since they used directly the scope window developed for SIMBIOS. Moreover, root locus, bode and Nyquist windows were easily implemented as subclasses of the scope window class.

- The incremental and evolutionary approach to software development supported by this methodology makes possible the continuous improvement of the SIMBIOS prototype. This was not possible in our C prototype since small conceptual changes could require a considerable effort to code them.

- Using object-oriented programming languages there is a substantial reduction in the size of the existing code. Therefore, a single person can manage more complexity (Roberts 1988). The reduction of the size of existing code has been particularly important in the implementation of the submodel, model and experiment instantiation mechanisms since the instantiation mechanisms are intrinsic to the object oriented methodology.

Since in Smalltalk is quite ease to create a prototype by using programming refinement. Programmers may be tempted to turn these first attempts into products without reexamining the code or the overall design. Often, a better strategy consists on developing one or more prototypes and then construct a new system reusing code from the prototype where convenient (Ewing 1987). SIMBIOS has had three previous prototypes. Figure 1 shows for each prototype its complexity versus its power. The prototype should be redesigned when it becomes difficult to grasp the interrelationship between its classes.

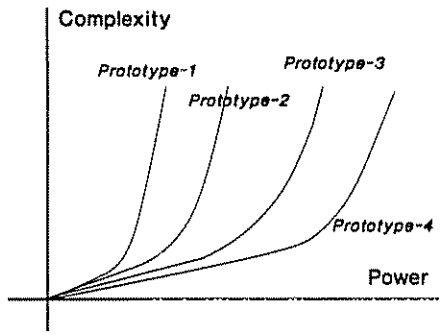

Figure 1. SIMBIOS prototypes

Although our programming environment of choice for fast prototyping is Smalltalk, it does not seem the right language for a final product since its simulation performance is very low compared against C++ (Dyke 1989; Doyle 1990).

THE SIMBIOS PROTOTYPE

Object oriented programming was first developed as a convenient approach to implement simulation problems. The notions of objects, classes, and message was introduced in the SIMULA language (Koschmann 1988). Object oriented simulation attempts to fill the gap between the model and what is modeled (Roberts 1988). In continuous model simulation, objects (submodels) are mathematical entities. However, in most cases, the objects are physical and observable (i.e. power plant).

The hierarchical relation among the SIMBIOS classes is shown in figure 8. SIMBIOS classes are written in reverse video. An exhaustive analysis of the class hierarchy is out of the scope of this paper. Therefore, only significant classes are introduced in the following sections.

Users interface

The simulation environment associates separate windows for each aspect of the work. For example, an icon-based window for the model definition, a window for the experiment specification and windows for graphical and numerical results. The users interface and in particular, the icon modeling window will support several modeling methodologies. For example, bond-graph, analog-like, control-like and systems theory. Therefore, the user will be able to choose the modeling formalism that suits him best.

The SIMBIOS modeling window (SimbiosIconPane) is icon-based. Each submodel has its own icon. A simulation is constructed by specifying icons and connecting them as required. The icon digraph (IconDigraph) holds topological information about the graphical model representation. SIMBIOS supports a hierarchical approach to modeling. A model may be composed of a number of submodels, each with its own icon. Clicking on the icon of the submodel opens up a detailed representation of the submodel.

Figure 2 represents the classical bouncing ball problem. Four submodels are needed to model it. The ground submodel is an instance of the DownWhen submodel class. An state event is detected when the ball reaches the ground. At event time, an impulse signal is sent to the speed submodel (ReseteableIntegrator class) and the input speed is latched and sent to the bouncingSpeed submodel (Gain class). Figure 3 shows the bouncing ball simulation results.

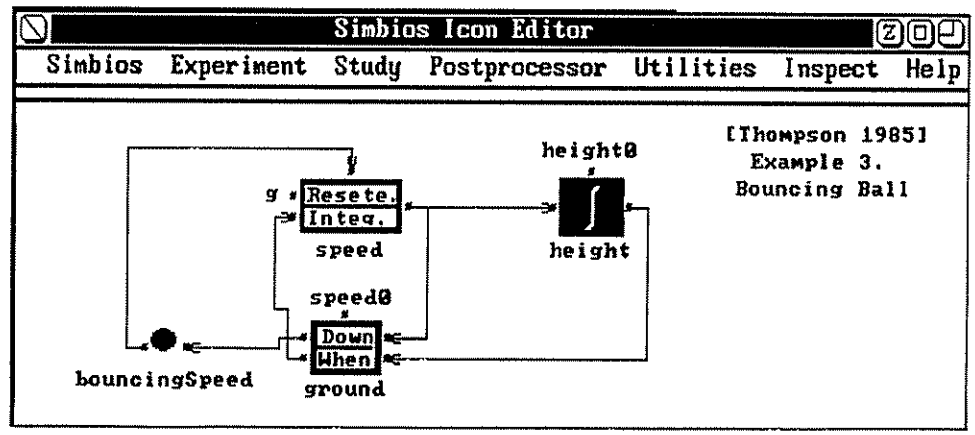

Figure 2. SIMBIOS icon modeling window

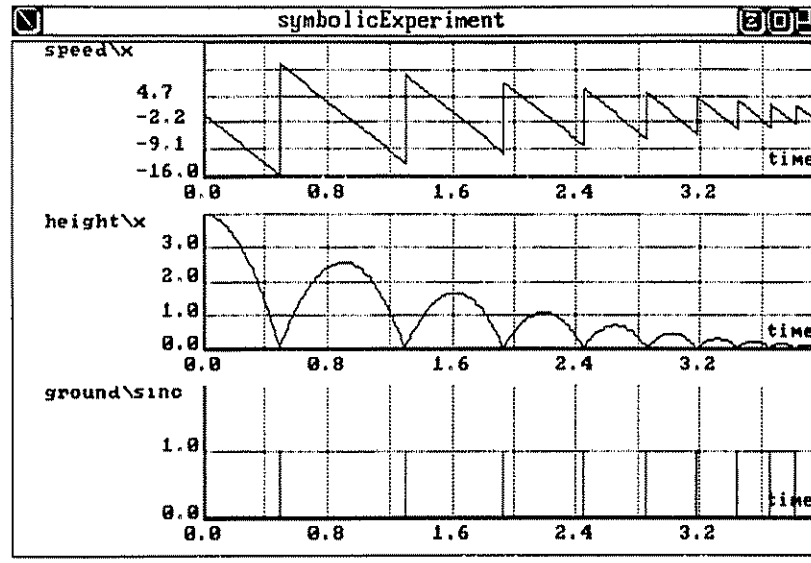

Figure 3. Scope window

The scope window (ScopeWindow class) is not a passive element; on the contrary, it has many useful features that can be activated with a mouse. Figure 7 shows three different capabilities: zooming, drawing a mark over each solution point and looking at the numerical value associated with any graphic point.

Experiment

The experiment (CompiledExperiment class) has been defined as an extension (subclass) of the generic submodel (Pcs) since it has the same structure plus a control segment and an output segment. The experiment is similar to the generic submodel class since it only holds experiment static characteristics. The procedural and declarative code specific to each experiment is defined in the experiment subclasses. Only compiled experiments are stored as subclasses of the CompiledExperiment class. Interactive experiments are instances of the SymbolicExperiment class.

The experiment window (ExperimentPane) is used to specify the experiment start time, end time, initial integration step, and the integration algorithm. A simulation clock is automatically created at the beginning of a simulation experiment and removed at the end of the experiment (figure 4).

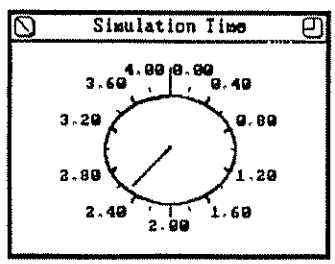

Figure 4. Simulation clock

Model

This SIMBIOS prototype supports only the modeling, simulation and experimentation of piecewise continuous models. A model is made of a hierarchical set of submodels. However, a model can be itself a submodel within a broader model. Therefore, a model is a relative concept which depends on the experiment being performed.

In SIMBIOS, large models may themselves be designated submodels and used in even larger simulations. The theoretical groundwork for piecewise continuous model and submodel internal representation and sorting is described in (Guasch 1987) Though, at present, only continuous models are handled, further extensions to discrete and combined simulation are expected.

SIMBIOS does not provide the basic CSSL program regions. However, its internal sorting algorithm does the same functionality. It splits the model into the following main segments: -initial, ode (derivative in CSSL languages), output (operations at communication intervals) and discontinuous segments. Figure 5 represents the ordered submodel message sequence for each bouncing ball model segment.

Each submodel message is represented as a vertex in a model digraph (ModelDigraph). The first vertex of the bouncing ball initial segment is an initial vertex (InitialVertex), the owner of this vertex is the height submodel, and the message name that initializes this submodel is x0:

The model digraph reflects the model computational flow and is directly used to perform the interactive experiments since it represents the sorting relationship between called submodel segments. Moreover, it is employed in the translation of the model into a submodel class which can later be reused as a submodel in a bigger model.

The way in which the user-supplied model is converted into executable code obviously affects the question of providing interaction. In continuous simulation the favoured approach is to use an interpretive language. It can be based on existing languages such as BASIC or it can be written from scratch (Crosbie 1982).

The SIMBIOS approach to interactive simulation differs notably from the previous ones. In classical interactive simulation languages the users model code is a static representation of the final executable model. Both model representations are related by a convertor which analyzes the users source code and translates it to executable code. On the contrary, in SIMBIOS the users model is made of active submodel instances. Therefore, no conversion is needed and a greater degree of flexibility can be achieved.

The SIMBIOS model is reordered and segmented before every simulation experiment.

```
INITIAL SEGMENT
Vertex(InitialVertex height x0:)
Vertex(StateVertex height x)
Vertex(InitialVertex ground control:)
Vertex(InitialVertex ground x0:)
Vertex(StateWhenVertex ground sinc)
Vertex(StateWhenVertex ground y)
Vertex(AlgebraicVertex bouncingSpeed x:y)
Vertex(InitialVertex speed x0:)
ODE SEGMENT
Vertex(StateWhenVertex ground sinc)
Vertex(StateWhenVertex ground y)
Vertex(AlgebraicVertex bouncingSpeed x:y)
Vertex(AlgebraicVertex speed x0:control:x)
Vertex(DerivativeVertex height dx:)
Vertex(DerivativeVertex speed dx:control:)
DISCONTINUOUS SEGMENT
Vertex(StateVertex height x)
Vertex(AlgebraicVertex bouncingSpeed x:y)
Vertex(AlgebraicVertex speed x0:control:x)
Vertex(DiscontinuousVertex ground control:x:)
OUTPUT SEGMENT
```

Figure 5. Bouncing ball code segments

Submodels

SIMBIOS has an minimal repertoire of linear, non-linear, algebraic, logic, interface and event-drive functions (figure 7). All the SIMBIOS submodels inherit from a generic submodel class (Pcs class) which holds the information and methods that are common to all the submodels. These methods access or modify the submodel static characteristics. Since each submodel has its own dynamic characteristics (initial region procedural code and dynamic region declarative code), the methods of each submodel classes, created as an extension of the submodel class, must contain the procedural and declarative code that models its particular behavior.

Classic block-oriented languages suffer from a lack of flexibility and from limits imposed on the number of blocks of different types which could be handled. These restrictions are and should be removed in the new simulation environments. SIMBIOS has no restrictions on the number of components. In SIMBIOS, the basic submodel library is equivalent to that of available CSSL or block oriented simulation languages. However, since submodels are implemented as objects, greater flexibility can be achieved. Moreover, submodel specialization can be done by inheritance.

A generic submodel window (PcsPane) has been defined to interact with submodel instances (figure 6). Specific windows have also been defined for several submodels. These windows are subclasses of the OperatorWindow class.

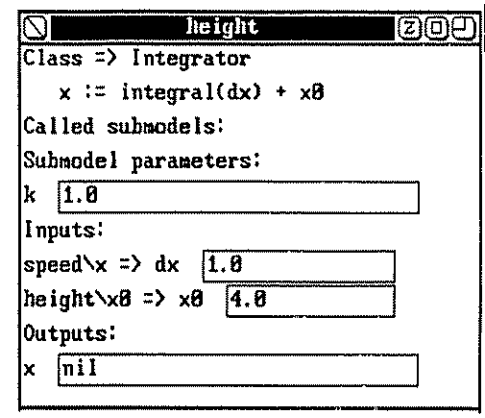

Figure 6. Generic submodel window

Engine

The simulation engine is the system component responsible for the execution of the simulation. After a SIMBIOS model has been specified by the user and before performing a simulation experiment, the environment sorts the submodel messages in order to execute them in the appropriate sequence. At simulation time, those messages and their associated input variables are explicitly sent to their associated submodel instances. This slows down the simulation process. However, a speed up of 50% can be achieved if we decide to compile the model translating it into a new class.

A generic integration (IntegrationAlgorithm class) has been defined in the SIMBIOS environment. It is a complete integrator except for the integration formula which is defined on its subclasses. Therefore, the generic submodel holds the mechanisms for output synchronization, and state and time event management. The generic integrator subclasses hold the method responsible of performing a simulation step.

SIMBIOS provides the most commonly used discontinuous operators. A discontinuity is detected by noting a change of sign in the value of a discontinuity function at the end of an integration step. The state event (StateEvent subclasses) proprietary of the discontinuity function uses linear and quadratic interpolation (Crosbie 1974) to predict the root time. The priority state event queue (StateEventQueue class) ensures that multiple discontinuities are dealt in the correct sequence. The state events in this priority queue are sorted out dynamically according to its expected root time. Once a discontinuity has been accurately detected, all the other state events than meet the error tolerances requirements are also taken care off.

SIMBIOS state event management mechanisms have been successfully tested using representative problems found in the literature (Thompson 1985; Birta 1985). Figure 7 is a scope window zoom that shows a bouncing ball discontinuity point.

CONCLUSIONS

SIMBIOS is an interactive icon-based continuous simulation environment. No translation or compilation is required. Therefore, interactive turnaround is very fast. SIMBIOS is currently implemented in Smalltalk/V 286 on IBM platforms. Smalltalk has been the language of choice because the dynamic binding and its programming environment ease the development of interactive program prototypes.

Our future goal is to enhance the capabilities of SIMBIOS by expanding the system, adding discrete event elements and including AI components (Luker 1989).

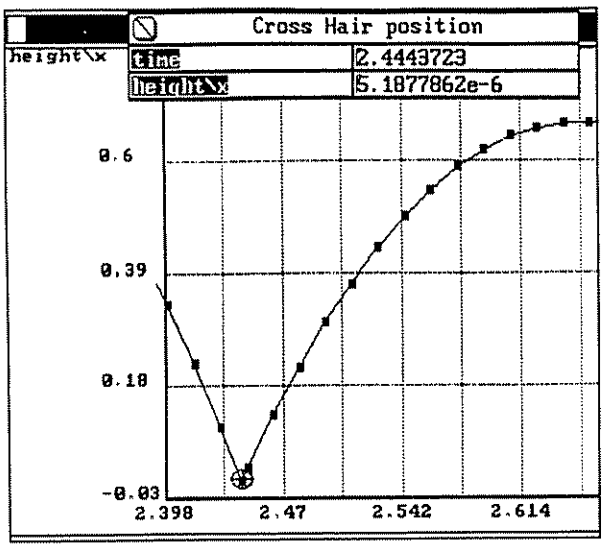

Figure 7. A scope window detail

REFERENCES

Birta L. G., Oren T. I. and Kettenis D. L. 1985. *"A Robust Procedure for Discontinuity Handling in Continuous System Simulation"* Transactions of the Society for Computer Simulation. Vol. 2 no. 3, 189-206 (Sept.).

Cox B. J. 1984. *"Object Oriented Programming Language. An evolutionary approach"*, Addison Wesley 10393, Reading, Massachusetts.

Crosbie R. E. and Hay J. L. 1974. *"Digital techniques for the Simulation of Discontinuities"*, Summer Computer Simulation Conference, Houston, Texas.

Crosbie R. E. 1982. *"Interactive and Real-Time Simulation"*, Progress in Modelling and Simulation, ed. Cellier F. E., Academic Press, 393-406.

Doyle J R. 1990. *"Object-Oriented Simulation Programming"* Object-Oriented Simulation, ed. Guasch A., SCS, 1-6.

Dyke R.P.T. and Kunz J.C. 1989. *"Object-Oriented Programming"*, IBM Systems Journal Vol. 28 no. 3, 465-478.

Ewing J. J. 1987. *"Smalltalk isn't meaningless chatter"*, Computer Design Vol. 15 (Jan.): 76-79.

Guasch A., Huber R. and Ilari J. 1984. *"ICDSL: Hacia una nueva definición de los lenguajes de simulación de sistemas continuos"*, 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. Universidad Politécnica de Cataluña. Barcelona - Spain (Dec.).

Guasch A. and Huntsinger R.C. 1989. *"Object-oriented continuous system simulation"* Summer Computer Simulation Conference. Austin, Texas (July).

Guasch A. and Luker, P. 1990. *"SIMBIOS: SIMulation Based on Icons and ObjectS"*. Object Oriented Simulation. The Society for Computer Simulation. Ed. A. Guasch. (Jan.):61-67.

Huber R. M. and Guasch A. 1986. *"Towards a specification of the structure for Continuous System Simulation Languages: Evolution and a proposal draft"* Vol. 2 in Computer Systems: Architecture, Applications, ed. M. Ruschitzka, North-Holland.

Koschmann T. and Walton Evens M. 1988. *"Bridging the Gap between Object-oriented and Logic Programming"*, IEEE Software (July):36-42.

Luker P. A. 1989. *"Intelligent Simulation Environments"*, Artificial Intelligence in Scientific Computation: Towards Second Generation Systems, ed. R. Huber et al, J.C. Baltzer AG. 15-25.

Meyer B. 1988. *"Object-oriented Software Construction"* Prentice-Hall. Series Ed. C.A.R. Hoare.

Oren T. I. 1975. *"Syntactic Errors of the Original Formal Definition of CSSL 1967"*, IEEE Computer Society Repository N. R75-78, Computer Science Dept., University of Ottawa., Ottawa (1975).

Pinson L.J. and Wiener R.S. 1988. *"An Introduction to Object-Oriented Programming and Smalltalk"* Addison Wesley Publishing Company.

Roberts D.S. and Heim J. 1988 *"A perspective on object-oriented simulation"*. Proceedings of the 1988 Winter Simulation Conference, 277-281.

Thompson S. 1985. *"Rootfinding and Interpolation with Runge-Kutta-Sarafyan Methods"*, Transactions of the Society for Computer Simulation Vol. 2 no. 3, (Sept.):207-218.

Wienner R.S. and Pinson L.J. 1988. *"An Introduction to Object-Oriented Programming and C+"*, Addison-Wesley Publishing Company, Reading, MA.