OBJECT ORIENTED CONTINUOUS SYSTEM SIMULATION

Antoni Guasch, Ph.D.
Institut de Cibernética
Polytechnic University of Catalonia
08028 Barcelona (Spain)

and

Ralph C. Huntsinger, Ph.D.
Department of Computer Science
Department of Mechanical Engineering
California State University, Chico
Chico, California 95929-0410

ABSTRACT

Continuous model simulation systems have evolved in the present decade from the monolithic CSSL 1967 architecture to new and more powerful modular ones. However, in spite of this constant evolution, the majority of the continuous model simulation systems available in the market have an architecture based on the 1967 standard. Furthermore, the programming language in which the majority of these systems are developed is still Fortran.

In spite of its engineering value, the use of Fortran or other classical procedural languages has constrained the degree of modularity, flexibility and reliability of the simulation systems. Therefore, the adoption of new programming methodologies is advisable. At present, it seems that the object-oriented approach is one of the most promising.

INTRODUCTION

The idea of modularization and submodels in Continuous System Simulation Languages (CSSL) has taken two approaches in the past. The first is the concept of a 'MACRO'. A MACRO is a piece of generic code that may be implemented and inserted into the general CSSL program. The parameters of the MACRO are used as symbol substitution strings to make each call of the MACRO a unique piece of inline code. Simulation languages with MACRO capability are CSSL-IV, ACSL and DSL/VS. The early versions of CSSL-III and CSSL-IV also used the *segment* concept. Submodels could be written using the segment operator and lined together at a later time as a single large FORTRAN program. Both of these approaches to modularization were constrained ultimately by the target FORTRAN code that has to be generated.

In continuous simulation a model or submodel is usually composed by a set of submodels. Furthermore, these submodels are, in general, closely related to physical subsystems. For example, a Proportional-Integral-Derivative (PID) controller, an Analog to Digital (A/D) converter or a Power Plant. Therefore, it seems that the object-oriented paradigm is suitable for representing the behavior of these physical subsystems. However, object oriented methodologies have been scarcely exploited in continuous simulation to represent the behavior of the continuous submodels.

The contribution of Object Oriented Methodologies and Languages to the simulation community has for a long time been recognized in the discrete simulation field. Note that SIMULA was designed to be employed in discrete simulation (Roberts 1988). Moreover, other object oriented languages have been designed or have been used extensively in discrete systems simulation.

Currently, it seems that previous restrictions can be removed using *Object-Oriented Methodologies* and *Languages*. Therefore, a run-time continuous system simulation environment prototype has been designed and coded using object-oriented principles . The same system has been coded and tested in Smalltalk/V 286 (Digitalk 1988) and in C++ (Stroustrup 1986). The first, offers the advantages of flexibility and a high level of problem abstraction while the second offers the advantages of execution speed and type consistency.

A SIMULATION ENVIRONMENT

Lets suppose that our run time simulation environment has the following elements:

- A set of *hierarchical models* where a hierarchical model is composed of a set of *submodels*. A submodel has an initial region plus a dynamic region. Furthermore, a submodel holds a set of *variables* and instances of the called lower level submodels.
- A set of *experiments*. An experiment monitors the execution of a single simulation run. An experiment may call zero, one or more hierarchical models. An experiment has the same structure of the submodel plus a terminal region, a control segment and an output segment. The control segment is intended for simulation run-time parameter specification such as initial time and finish time. The output segment clusters the output control statements. Its objective is to make explicit the variables whose evolution has to be recorded.
- A *Runge Kutta Fehlberg integration algorithm*.
- A *printer controller* responsible for printing a set of specified variables at periodic intervals.
- A *scope controller* responsible for drawing the dynamic behavior of a set of variables at periodic intervals.
- *Time events* used to print or draw simulation results at periodic intervals.

Moreover, this run-time simulation environment is going to be used for studying the *Non linear system* model represented in figure 1. It has two submodels: a *real pole* and a *limiter*. Furthermore in addition to perform a desired final experiment (*NonLinearSystemExperiment*) over the non linear system it is also advisable to test if the behavior of both submodels are correct. Thus, two more experiments have been included in the system (*LimiterTest* and *RealPoleTest*)

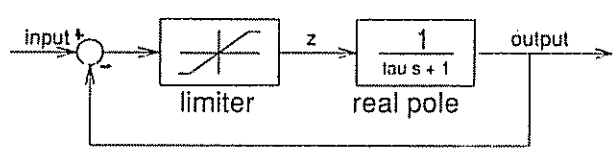

Figure 1: *Non linear system*.

Figure 2 represents the explicit relationship between the submodels of the non linear system. These submodels are connected through explicit calls to the lower level submodels.

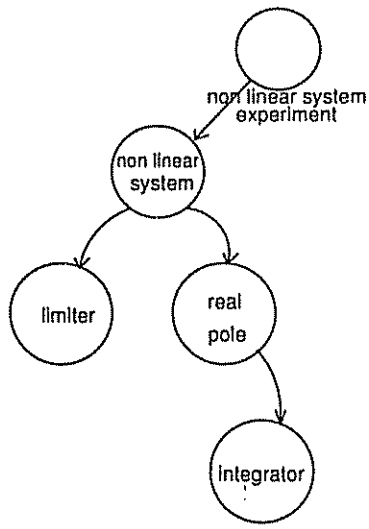

Figure 2: *Non linear system model hierarchy*

OBJECT-ORIENTED DESIGN

The hierarchical relation among the classes defined for this prototype is shown in listing 1. Only the emphasized classes have been implemented and tested. Moreover, this is the minimum set of classes needed for supporting the proposed study. Other classes can be directly added to support, for example, integer, logical, matrix and other type of operations.

```
Submodel
  Limiter
  RealPole
  Integrator
  NonLinearSystem
  ::::::::::
  Experiment
    LimiterTest
    RealPoleTest
    NonLinearSystemExperiment
    :::::::
Variable
  Constant
  StateVariable
    RKFStateVariable
Event
  TimeEvent
    ModelTimeEvent
    OutputTimeEvent
  StateEvent
    IfStateEvent
    WhenStateEvent
IntegrationAlgorithm
  OneStepMethod
    EulerMethod
    RungeKuttaMethod
      SecondOrderRungeKutta
      ThirdOrderRungeKutta
      ::::::::::::::::::
  AdaptiveRungeKutta
    RungeKuttaFehlberg
  ::::::
OutputController
  PrinterController
  ScopeController
  PrepareController
```

Listing 1: *Simulation environment hierarchy*

Object-Oriented Programming languages support the following properties (Pinson 1988): *data abstraction, encapsulation, inheritance* and *polymorphism*. Each one of theses properties can contribute to increase the flexibility and modularity of the simulation environment.

The above concepts have been widely examined in the last few years. Therefore, in this study we are only going to focus on the role that these concepts can play in the design of a run-time continuous simulation environment.

Data Abstraction and Encapsulation

An abstract data type is a model that encompasses a type and an associated set of operations. These operations are defined for and characterize the behavior of the underlying type (Wiener 1988). Moreover, a data abstraction hides the details of the way in which data are represented. Abstraction is supported by object-oriented programming languages since objects are encapsulations of abstractions (Pinson 1988). In object-oriented programming new data abstractions can be specified through the definition of new classes.

In object-oriented programming, it is normal to define new classes using existing ones. For example, if the *RealPole* and the *Limiter* classes have been defined, a new class *NonLinearSystem* can be defined specifying:

- the private data for the new class. The data may be instances of existing classes. In C++,

```
RealPole plant;
Limiter op;
Variable lowerLimit, higherLimit,tau;
Variable input, output, error, z;
```

- the methods (member functions in C++) of the new class. They manipulate the internal data of the object. This submodel as well as the rest of the submodels of the system have a set of methods that are consistent with the functional tasks involved in a simulation run: initialization, derivative computations and discontinuous function computations (Guasch 1987).

Encapsulation is a technique for minimizing interdependencies among separately-written modules by defining strict external interfaces (Micallef 1988). Implementation details are hidden inside the module. Thus, it is possible to change the module without having to change or recompile the rest of the application.

The structure of classic CSSL67 languages does not support the encapsulation idea. As a consequence, a change in the data structure or code of a simulation operator may imply, not only recompilation of almost all the simulation library subprograms, but changes in its code (in general, COMMON declarations). The cause of this problem was the use of a general COMMON to store the memory data, the data needed for information management (symbolic access) and the state, derivative vectors and auxiliary vectors.

If we want to support the encapsulation concept in this system, the communication between modules has to be done through clear interfaces. For example, it is the responsibility of the *submodel* to answer or modify the value of an internal *parameter*, to reply the

called lower level submodels and to add in the state vector of the *integration algorithm* the submodel *state variables*. Therefore, classical approaches, based on the use of COMMON, to support symbolic access and vector integration must be and can be removed.

Information management is not a critical aspect in terms of computation time since related operations are always performed before or after a simulation *experiment*. But, other time critical aspects such as integration can be less efficient as a result of strictly applying the encapsulation concept. However, in programming languages with object oriented features such as C++, strict encapsulation can be bypassed in order to reduce the time overhead in critical modules with a minimum loss of flexibility.

Inheritance

Inheritance is the property that allows a class to be defined as an extension or as a restriction of another (Meyer 1988). This important object-oriented feature has been extensively used in the design of the run-time simulation environment (listing 1.).

The *Submodel* class holds the information and methods that are common to all the submodels. The *Submodel* 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 the *Limiter*, *RealPole* and *NonLinarSystem* classes, created as an extension of the *Submodel* class, must contain the submodel procedural and declarative code that models its particular behavior. This code is distributed into a set of methods in order to avoid information loops in the submodel declarative code (Guasch 1987).

Furthermore, the *Experiment* class can be defined as an extension of the *Submodel* class since it has the same structure plus a terminal region, a control segment and an output segment. Moreover, the *Experiment* class includes variables that control the simulation run (experiment) and methods that operate over them, some experiment variables are, for example, start time, end time, initial integration step and an instance of the *Runge Kutta Fehlberg* integration algorithm. The *Experiment class* is similar to the *Submodel* class since it only holds experiment static characteristics. The procedural and declarative code specific to each experiment is defined in the *Experiment* subclasses *LimiterTest*, *RealPoleTest* and *NonLinearSystemExperiment*.

Listing 1 shows the hierarchical relationship between the classes defined for this simulation environment. One can observe in it the importance of the inheritance concept in this implementation.

Polymorphism

While inheritance address the problem of reusability (building modules as extensions of existing ones, polymorphism faces the problem of extensibility. Polymorphism is an important feature of all object-oriented languages that allows the definition of flexible software elements amenable to extension and reuse. A polymorphic operator (method) is one that has multiple meanings depending on the type (class) of its arguments (Micallef 1988).

This property plays an important role in our prototype since it allows an easy extension of it and also makes it more flexible. The use of this property in the Smalltalk environment is straightforward. However, as it will be shown in the next paragraphs, the hierarchical relation between the classes represented in listing 1. have to be modified in the C++ prototype because polymorphism is constrained by inheritance, in this typed language.

Polymorphism is a key concept in solving the information management and symbolic access aspects of the hierarchical run-time simulation system. For example, it allows the inclusion of references to submodel *variables* of different types and of references to lower level submodels in the *submodel* symbol table.

Furthermore, polymorphism allows us to send the same message to the different elements of the symbol table and have each element respond in a way appropriate to its type. For example, if we want to know the symbolic name of a given element of the table we can simply send the message name to it without having to worry about the particular type of the element.

For example, the *non linear system experiment* represented in figure 2. has a symbol table and one of the elements of the table is a reference to the *non linear system* submodel. The symbol table of this submodel contains references to the submodel variables (tau, upperLimit, lowerLimit,...) and references to the lower level submodels. This can be extended to all the submodels of the hierarchy. Therefore, if the *submodel* class has a method called extractElement which looks for a given element into the submodel or experiment symbol table and returns it, then the next Smalltalk code modifies the value of a *non linear system* internal *variable* (parameter),

```
experiment := NonLinearSystemExperiment new.
submodel := experiment extractElement:
   'nonLinerSystem'.
(submodel extractElement: 'tau') setTo: 2.0.
```

the C++ equivalent code is,

```
NonLinearSystemExperiment experiment;
submodel := experiment.extractElement
      ("nonLinerSystem");
submodel->extractElement("tau")->setTo(2.0);
```

Moreover, the extractElement method can be improved to allow 'recursive' access to the hierarchical models. Therefore, the Smalltalk code equivalent to the previous one is,

```
experiment := NonLinearSystemExperiment new.
(experiment extractElement: 'nonLinerSystem\tau')
            setTo: 2.0.
```

and the C++ code is,

```
NonLinearSystemExperiment experiment;
experiment.extractElement("nonLinerSystem\tau")
          ->setTo(2.0);
```

Since polymorphic vectors are constrained in C++ by inheritance, a common parent for the *Submodel* and *Variable* classes has to be defined in the C++ prototype since the symbol table holds references to submodels as well as to variables.

Another good example of polymorphism can be found in the management of time events. Two types of time events can be found in this prototype: *model time events* that have their origin in the model (i.e. to control the sampling intervals in a digital controller); and *output time events* that are generated by the *output controllers* (i.e. to print the dynamic results at periodic times). Both events are sent to the same priority queue owned by the *integration algorithm* and it has the responsibility to execute the actions associated to each event. The Smalltalk code to perform the event action is,

```
timeEventQueue firstElement owner perform
```

and the C++ equivalent code is,

```
timeEventQueue->firstElement()->owner()->perform();
```

the message firstElement is sent to the timeEventQueue and the result of this operation is an *event*. Afterwards, the message owner is sent to the *event* and the result is the owner of the event (a *submodel*, a *printer controller* or a *scope controller*.). Moreover, the message *perform* is sent to the owner of the event in order to

execute the action associated to it. Therefore, if the owner of the event is a *printer controller*, the simulation results will be printed and if the owner is a submodel, it will be called to perform the action associated to its event. Notice that, though the *modelTimeEvent* class and the *prepareController* class have not been included in the system, their inclusion is straightforward thanks to the polymorphism property. As a final remark, the C++ polymorphic constraints also affects the implementation of this aspect of the system.

CONCLUSIONS

A run-time continuous system simulation environment has been designed, coded and tested in two different languages with object-oriented capabilities: Smalltalk and C++. The untyped, dynamic binding and programming environment of Smalltalk makes it an excellent language for prototyping, whereas C++ is faster but does not have, at present, a programming environment. However, in both prototypes, the code is more reliable and flexible than a previous one implemented in C thanks to the encapsulation, data abstraction, inheritance and polymorphism properties of both languages. Therefore, it seems that object-oriented methodologies will play an important role in continuous system simulation.

ACKNOWLEDGMENTS

This work has been made possible thanks to the support obtained from a research grant from the 'Ministerio de Educación y Ciencia' of Spain and from the Fulbright institution.

REFERENCES

Chapra S.C. and Canale R.P. 1988. *Numerical Methods for Engineers*, McGraw-Hill Inc.

Digitalk Inc. 1988. *Smalltalk/V 286 - Tutorial and Programming Handbook*. Digitalk Inc.

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

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

Micallef J. 1988. "Encapsulation, Reusability and Extensibility in Object-Oriented Programming Languages". *Journal of Object-Oriented Programming*, Vol 1 (1), pp 12-35 (April/May 1988).

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*, Abrams M., Haigh P. and Comfort J. (eds.), pp. 277-281.

Stroustrup B. (1986). *The C++ Programming languages* Addison Wesley Publishing Company.

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